An Interface Theory for Service-Oriented Design

José Luiz Fiadeiro[a], Antónia Lopes[b]

[a]*Department of Computer Science, Royal Holloway University of London, UK*
[b]*Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal*

Abstract

We put forward an interface and component algebra through which we characterise fundamental structures that support service-oriented design independently of the specific formalisms that may be adopted to provide models for languages or analysis tools. We view services as an interface mechanism that can be superposed over a component infrastructure, what is sometimes referred to as a 'service overlay'. The component algebra consists of networks of processes that interact asynchronously through communication channels. A service interface offers properties to potential clients and requires properties of external services that, at run time, may need to be discovered and bound to the orchestration of the service. We define what it means for an asynchronous relational net to orchestrate a service interface and prove a number of compositionally results that relate the operations of both algebras. One of the major results of the paper is the characterisation of a sub-class of asynchronous relational nets over which we can guarantee that, when binding, through their interfaces, a client and a supplier service, the composition of the orchestrations of the two services is consistent, i.e., both services can work together as interconnected.

Keywords: asynchronous process networks, component algebra, interface algebra, orchestration, service-oriented computing, temporal logic

1. Introduction

In [18], de Alfaro and Henzinger put forward a number of important insights, backed up by mathematical models, that led to an abstract characterisation of essential aspects of component-based software design (CBD), namely the distinction between the notions of component and interface, and the way they relate to each other. This separation is important because component and interface models capture different aspects of software system engineering: the former characterise the way components behave in an arbitrary environment and support verification; the latter characterise what components offer to and expect from the environment, and support design.

In this paper, we take stock of the work that we developed in the FET-GC2 integrated project SENSORIA [55] around a language (SRML) and mathematical model for service-oriented modelling [33], and investigate what abstractions can be put forward for service-oriented computing (SOC) that relate to the notions of interface and component algebra proposed in [18]. Our ultimate goal is to characterise the fundamental structures that support SOC independently of the specific formalisms (Petri-nets, different kinds of automata or state machines, process calculi, *inter alia*) that may be adopted to provide models for languages or analysis tools.

This leeds us to explore (Sect. 1.1) the difference between CBD and SOC, which is not uncontroversial. We view services as an interface mechanism that can be superposed over a component infrastructure, what is sometimes referred to as a 'service overlay'. Our interface theory formalises this view, which results in component and interface algebras that differ in fundamental points from those proposed in [18]. They also differ substantially from other notions of web-service interface that have been proposed in the literature, e.g., [9], in that our notion of composition is not for integration (as in CBD) but for dynamic interconnection of processes: the interfaces used in [9] are meant for design-time composition, the client being statically bound to the invoked service (which is the same for all invocations); the interfaces that we proposed address a different form of composition in which the provider is procured at run time and, therefore, can differ from one invocation to the next. This difference is also reflected in the fact that our component algebra consists of asynchronous networks of processes (distributed orchestrations), not individual processes. Therefore, our first contribution is in providing an interface and a component algebra that better reflect SOC as an engineering paradigm, for example the Service Component Architecture (SCA) [47].

In relation to SRML and our previous work in SENSORIA (e.g., [30, 31, 33]), this paper separates the component from the interface algebra (the notion of module in SRML corresponds to an orchestrated interface), which allows us to address aspects such as consistency, abstraction and refinement, which are essential for the toolbox of any software engineer. Our second contribution is, therefore, the formalisation of those aspects as applied to SOC, especially consistency: one of the major results of the paper is the characterisation of a sub-class of asynchronous networks of processes over which we can guarantee that, when binding, through their interfaces, a client and a supplier service, the composition of the orchestrations of the two services is consistent, i.e., both services can work together as interconnected. That is, consistency can be checked at design time, not at the time of binding (which in SOC is done at run time). Once again, this captures a major difference between CBD and SOC.

1.1. Services vs. components, informally

A question that, in this context, cannot be avoided, concerns the difference between component-based and service-oriented design. The view that we adopt herein is that, on the one hand, services offer a layer of activity that can be superposed over a component infrastructure (what is sometimes referred to as a 'service overlay') and, on the other hand, the nature of the interactions that are needed to support such a service overlay is intrinsically asynchronous and conversational, which requires a notion of component algebra that is different from the ones investigated in [18] for CBD.

The difference between components and services, in what relates to the purpose of this paper, can be explained in terms of two different notions of 'composition', requiring two different notions of interface. In CBD, composition is integration-oriented — "the idea of component-based development is to industrialise the software development process by producing software applications by assembling prefabricated software components" [22]. In other words, CBD addresses what, in [25] we have called 'physiological complexity' — the ability to build a complex system by *integrating* a number of independently developed parts. Hence, interfaces for component-based design must describe the means through which software elements can be *plugged together* to build a product and the assumptions made by each element on the environment in which it will be deployed. Interfaces in the sense of [18] – such as assume/guarantee interfaces – fall into this category: they specify the combinations of input values that components

implementing an interface must accept from their environment (*assumptions*) and the combinations of output values that the environment can expect from them (*guarantees*).

In contrast, services respond to the necessity for separating "need from the need-fulfilment mechanism" [22] and address what in [25] we have called 'social complexity': the ability of software elements to engage with other parties to pursue a given business goal. For example, we can design a seller application that may need to use an external supplier service if the local stock is low (*the need*); the discovery and selection of, and binding to, a specific supplier (*the need-fulfilment mechanism*) are not part of the design of the seller but performed, at run time, by the underlying middleware (service-oriented architecture) according to quality-of-service constraints. In this context, service interfaces must describe the properties that are provided (so that the services offered by applications can be discovered) as well as those that may be required from external services (so that the middleware can select a proper provider). The latter are not assumptions on the environment as in CBD — in a sense, through run-time discovery and binding, applications *create* the environment in which they need to operate in order to deliver the services that they promise (a form of dynamic reconfiguration that we detail in [29]).

This difference has implications on the nature and properties of the algebras that capture the operations on and relationships between design elements. For example, in CBD, composition is commutative, reflecting that we are building something more complex from simpler parts: software applications are composed as components of a bigger whole; the process stops when the designer has assembled all the components it needs. In SOC, composition is not commutative, reflecting the fact that it supports the binding of a client to a supplier: the purpose is not to build a bigger whole but to bind an application to the external suppliers that it needs to deliver a service. The process continues if some of those suppliers, in turn, require external services, and so on, leading to a dynamic form of composition that results in a configuration that cannot be predicted at design time.

Concerning the key questions that [18] identifies for distinguishing between components and interfaces, where the question *What does it do?* in the context of describing components that communicate synchronously through I/O ports ultimately means *How are inputs transformed into outputs?*, in SOC it should mean *How are interactions orchestrated among a group of partners?*; and where the question *How can it be used?* in the context of describing component interfaces ultimately means *What constraints apply to the values that can be passed through the ports?*, in SOC it should mean *What are the protocols that parties need to observe at the ports to engage with the service?*. These differences have important methodological and design implications as decisions need to be made about whether coupling should be tight or loose, binding should be static (at design time) or dynamic (at run time), communication should be synchronous or asynchronous, and so on.

1.2. Overview of the paper

In the context of modelling and specifying services, one can find approaches of two different kinds — choreography and orchestration — which are also reflected in the languages and standards that have been proposed for Web services, namely WS-CDL for choreography and WS-BPEL for orchestration. In a nutshell, choreography is concerned with the specification and realizability of a 'conversation' among a (fixed) number of peers that communicate with each other to deliver a service, whereas orchestration is concerned with the definition of a (possibly distributed) business process

(or workflow) that may use external services discovered and bound to the process at run time in order to deliver a service.

Whereas the majority of formal frameworks that have been developed for SOC address choreography (see [51] for an overview), the approach that we take in this paper is orchestration-oriented. More precisely, we propose to model the workflow through which a service is orchestrated as being executed by a network of processes that interact asynchronously and offer interaction-points to which clients and external services (executed by their own networks) can bind. The questions that we propose to answer are *What is a suitable notion of interface for such asynchronous networks of processes that deliver a service?*, and *What notion of interface composition is suitable for the loose coupling of the business processes that orchestrate the interfaces?*

The rest of this paper is technical and formal: our purpose is not to define an interface language but, rather, characterise the fundamental structures that support software engineering for SOC and the way it differs from CBD. Methodological aspects of our approach can be found in previous publications, e.g., [1, 10, 11, 33].

In Section 2, we present a 'component algebra' that builds on networks of processes that communicate asynchronously, i.e., components are networks. We define a law of composition and an abstraction mechanism that we prove to be compositional. A significant part of this section is dedicated to the characterisation of a subclass of asynchronous relational nets that are guaranteed to be consistent (i.e., they admit at least one run that projects to valid runs of the processes in the network and the channels through which they communicate) and closed under composition. This characterisation is given in terms of properties that can be checked at design time; checking for consistency at discovery time would not be credible because, in SOC, there is no time for the traditional design-time integration and validation activities as the SOA middleware brokers need to discover and bind services at run time.

This leads us to the characterisation of services as an 'interface algebra' (again in the sense of [18]), which we develop in Section 3. Interfaces involve the specification of behaviour in terms of temporal logic. More precisely, an interface consists of a provides-point through which properties that are offered to clients of the service can be specified, and a collection of requires-points through which properties can be specified that are required of the external services that, at run time, may need to be discovered and of the channels through which they will bind to the orchestration of the service. We define a law of composition and a refinement mechanism that we prove to be compositional.

Figure 1 summarises the formalisms used in the paper. The interface algebra builds on specification logics (Sect. 3.1) – a form of institutions [36], an example of which is SAFETY-LTL (Sect. 3.5), a version of Parametric Temporal Logic – PLTL [4] where intervals are finite and bounded by constants. The algebra includes operations of composition (Sect. 3.3) and refinement (Sect. 3.4). The component algebra builds on the topological space of infinite traces (Sect. 2.1), and includes operations of composition (Sect. 2.4) and abstraction (Sect. 2.5).

The two algebras are connected by defining what it means for an asynchronous relational net to orchestrate a service interface (Sect. 3.2). The notion of orchestration is based on the satisfaction relation of the specification logics, which relates sets of traces with sentences. We prove a number of compositionality results that relate the operations of both algebras, which use the algebraic properties of specification logics as institutions. Those results allow us to formulate properties that guarantee that, when binding, through their interfaces, a client and a supplier service, the composition of the orchestrations of the two services is consistent, i.e., both services can work together as

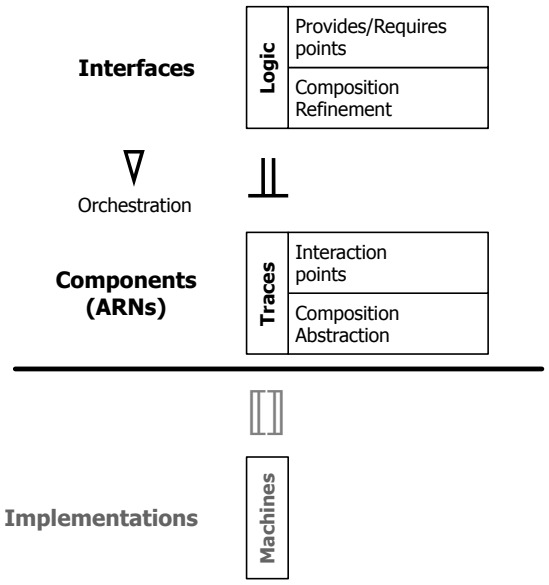

Figure 1: Summary of the formalisms used in the paper.

interconnected.

A third formalism accounts for the means through which components can be actually implemented. Typically, this involves some notion of 'machine' that generates sets of traces, an example of which are Büchi automata (Sect. 2.4). In this paper, we do not develop a corresponding 'machine algebra', which could build, for example, on operations of composition and simulation over automata. The advantage of building the component algebra over sets of traces is that it makes our results independent of the specific choice of machine, which in the literature on services include, for example, Petri-nets [49] and guarded automata [34] as abstractions of web services orchestrated in BPEL (the Business Process Execution Language [52]) .

In Section 4, we compare our framework with formal models that have been proposed in the last few years for SOC, including [7, 9, 35]. Finally, we conclude with a summary of the results obtained in the paper and a discussion of further work.

The paper builds on two of our previous papers ([27, 28]) but extends them in a significant way. In relation to [27], the component algebra has been changed to that of [28] in order to use the topological properties that are required to address consistency of composition. A notion of component abstraction has also been added, which had not been previously defined. Sect. 3 is almost totally new. On the one hand, the notion of orchestration proposed in [27] had to be redefined in order to take into account the revised component algebra and build on the new notion of component abstraction. On the other hand, a notion of interface refinement was introduced, for which several compositionality results had to be proved.

2. The component algebra

In this paper, we adopt the view that services are delivered by systems of components as in the Service Component Architecture (SCA) [47]:

> "SCA is a model designed for SOA, unlike existing systems that have been adapted to SOA. SCA enables encapsulating or adapting existing applications and data using an SOA abstraction. SCA builds on service encapsulation to take into account the unique needs associated with the assembly of networks of heterogeneous services.
>
> SCA provides the means to compose assets, which have been implemented using a variety of technologies using SOA. The SCA composition becomes a service, which can be accessed and reused in a uniform manner. In addition, the composite service itself can be composed with other services [...]. SCA service components can be built with a variety of technologies such as EJBs, Spring beans and CORBA components, and with programming languages including Java, PHP and C++ [...]. SCA components can also be connected by a variety of bindings such as WSDL/SOAP web services, Java™ Message Service (JMS) for message-oriented middleware systems and J2EE™ Connector Architecture (JCA)".

In the terminology of [18], we can see components in the sense of SCA as implementing *processes* that are connected by *channels*. However, there is a major difference in the way processes are connected. In [18], and indeed many models used for service choreography and orchestration (e.g., [7, 16, 49]), communication is synchronous (based on I/O connections). In order to capture the forms of loose coupling that SOAs support, communication should be asynchronous: in most business scenarios, the traditional synchronous call-and-return style of interaction is simply not appropriate. This leads us to propose a model that is closer to communicating finite-state machines [13] (also adopted in [8]) than, say, I/O automata [42]. We call our (service) component algebra *asynchronous relational nets* (ARNs).

2.1. Preliminaries

The processes that execute in SOC are typically open, reactive and interactive. Their behaviour can be observed in terms of the actions that they perform. For simplicity, we use a linear time model, i.e., we observe streams of actions. In order not to constrain the environment in which processes execute and communicate, we take streams that capture complete behaviours to be infinite (which we call traces) and we allow several actions to occur 'simultaneously', i.e., the granularity of observations may not be so fine that we can always tell which of two actions occurred first. The execution of an empty set of actions corresponds to a step during which a process is idle, i.e., a step performed by the environment without the involvement of the process.

The following definition sets out terminology and notation that is used throughout the paper.

Definition 2.1 (Trace, segment, and property)**.** *Let A be a finite set (of actions).*

- *A* trace λ *over A is an element of $(2^A)^\omega$, i.e., an infinite sequence of sets of actions. We denote by $\lambda(i)$ the (i+1)-th element of λ, by λ_i the prefix of λ that ends at $\lambda(i)$, and by λ^i the suffix of λ that starts at $\lambda(i)$.*
- *A* segment *over A is an element of $(2^A)^*$, i.e., a finite sequence of sets of actions. We use $\pi \prec \lambda$ to mean that the segment π is a prefix of λ.*
- *Given $A' \subseteq A$ and a segment π, we denote by $(\pi \cdot A')$ the segment obtained by adding A' at the end of π. We use the same symbol to denote the concatenation of segments ($\pi_1 \cdot \pi_2$) and of segments with traces ($\pi \cdot \lambda$).*

- *A* property Λ *over* A *is a subset of* $(2^A)^\omega$.

Notice that finite behaviours can be captured through traces that, after some point, consist only of the empty set, i.e., they are of the form $\pi.\emptyset^\omega$ where $\pi \in (2^A)^*$.

Definition 2.2 (Closure)**.** *Let* A *be a set and* Λ *a property over* A*. We define:*

- $\Lambda^f = \{\pi \in (2^A)^* : \exists \lambda \in \Lambda(\pi \prec \lambda)\}$ *— the segments that are prefixes of traces in* Λ*, also called the* downward closure *of* Λ.
- $\bar{\Lambda} = \{\lambda \in (2^A)^\omega : \forall \pi \prec \lambda(\pi \in \Lambda^f)\}$ *— the traces whose prefixes are in* Λ^f*, also called the* closure *of* Λ.
- Λ *is said to be* closed *iff* $\Lambda \supseteq \bar{\Lambda}$ *(and, hence,* $\Lambda = \bar{\Lambda}$*).*

The closure operator on $(2^A)^\omega$ is defined according to the Cantor topology used in [2] for characterising safety and liveness properties (see also [6]). In that topology, the closed sets are the safety properties (and the dense ones are the liveness properties).

Functions between sets of actions, which we call *alphabet maps*, are useful for defining relationships between individual processes and the networks in which they operate.

Definition 2.3 (Projection and translation)**.** *Let* $\sigma : A \to B$ *be a function (alphabet map).*

- *For every* $\lambda' \in (2^B)^\omega$*, we define* $\lambda'|_\sigma \in (2^A)^\omega$ *pointwise as* $\lambda'|_\sigma(i) = \sigma^{-1}(\lambda'(i))$ *— the* projection *of* λ' *over* A*. If* σ *is an inclusion, i.e.,* $A \subseteq B$*, then we tend to write* $\_|_A$ *instead of* $\_|_\sigma$*; this is a function that, when applied to a trace, forgets the actions of* B *that are not in* A*.*
- *For every property* $\Lambda \subseteq (2^A)^\omega$*, we define* $\sigma(\Lambda) = \{\lambda' \in (2^B)^\omega : \lambda'|_\sigma \in \Lambda\}$ *— the* translation *of* Λ *to* B*.*

We are particularly interested in translations defined by prefixing every element of a set with a given symbol. Such translations are useful for identifying in a network the process to which an action belongs — we do not assume that processes have mutually disjoint alphabets. More precisely, given a set A and a symbol p, we denote by $(p.\_)$ the function that prefixes the elements of A with '$p.$'. Note that prefixing defines a bijection between A and its image $(p.A)$.

Alphabet maps induce translations that preserve closed properties:

Proposition 2.4 (Translation)**.** *Let* $\sigma : A \to B$ *be an alphabet map. For every closed property* Λ *over* A*,* $\sigma(\Lambda)$ *is a closed property over* B*.*

Proof. This is a simple property of the topological space defined by traces. □

2.2. Asynchronous relational nets

In an asynchronous communication model, interactions are based on the exchange of messages that are transmitted through channels (wires in the terminology of SCA). For simplicity, we ignore the data that messages may carry. We organise messages in sets that we call *ports*. More specifically, every process consists of a (finite) collection of mutually disjoint ports, i.e., each message that a process can exchange belongs to exactly one of its ports. Ports are communication abstractions that are convenient for organising networks of processes as formalised below.

Every message belonging to a port has an associated *polarity*: $^-$ if it is an outgoing message (published at the port) and $^+$ if it is incoming (delivered at the port). This is the notation proposed in [13] and also adopted in [7].

Definition 2.5 (Port and message polarity)**.** *A port is a finite set (of messages). Every port M has a partition $M^- \cup M^+$. The messages in M^- are said to have polarity $^-$, and those in M^+ have polarity $^+$.*

The actions of sending (publishing) or receiving (being delivered) a message m are denoted by $m!$ and $m¡$, respectively.

Definition 2.6 (Action)**.** *Let M be a port and $m{\in}M$.*

- *The set of actions associated with M^- is $A_{M^-} = \{m! : m{\in}M^-\}$.*
- *The set of actions associated with M^+ is $A_{M^+} = \{m¡ : m{\in}M^+\}$.*
- *The set of actions associated with M is $A_M = A_{M^-} \cup A_{M^+}$.*

A process is a non-empty property over the alphabet generated from a finite set of mutually disjoint ports:

Definition 2.7 (Process)**.** *A process consists of:*

- *A finite set γ of mutually disjoint ports.*
- *A non-empty property Λ over $A_\gamma = \bigcup_{M\in\gamma} A_M$.*

We also define $A^+_\gamma = \bigcup_{M\in\gamma} A_{M^+}$ and $A^-_\gamma = \bigcup_{M\in\gamma} A_{M^-}$.

Interactions are established through channels. Channels transmit messages both ways, i.e., they are bidirectional, which is consistent with [13]. Notice that, in some formalisms (e.g., [8]), channels are unidirectional, which is not so convenient for capturing typical forms of conversation that, like in SCA, are two-way: a request sent by the sender through a wire has a reply sent by the receiver through the same wire (channel). This means that channels are agnostic in what concerns the polarity of messages: these are only meaningful within ports.

Definition 2.8 (Channel)**.** *A channel consists of:*

- *A set M of messages.*
- *A non-empty property Λ over the alphabet $A_M = \{m!, m¡ : m \in M\}$.*

We also define $A^+_M = \{m¡ : m{\in}M\}$ and $A^-_M = \{m! : m{\in}M\}$.

Notice that in [8] as well as other asynchronous communication models adopted for choreography, when sent, messages are inserted in the queue of the consumer. In the context of loose coupling that is of interest for SOC, channels (wires) may have a behaviour of their own that one may wish to describe or, in the context of interfaces, specify. Therefore, for generality, we take channels as first-class entities that are responsible for delivering messages and, hence, may have their own buffers. More specifically, we consider that the publication of messages are inputs for channels and the delivery of messages are inputs for processes.

Channels connect processes through their ports. Formally, the connections are established through what we call attachments:

Definition 2.9 (Connection)**.** *Let M_1 and M_2 be ports and $\langle M, \Lambda\rangle$ a channel. A connection between M_1 and M_2 via $\langle M, \Lambda\rangle$ consists of a pair of injections $\mu_i{:}M{\to}M_i$ such that $\mu_i^{-1}(M_i^+) = \mu_j^{-1}(M_j^-)$, $\{i, j\}{=}\{1, 2\}$. Each injection μ_i is called the* attachment *of $\langle M, \Lambda\rangle$ to M_i. We denote the connection by $\langle M_1 \overset{\mu_1}{\leftarrow} M \overset{\mu_2}{\rightarrow} M_2, \Lambda\rangle$.*

A connection establishes a one-to-one correspondence between the two ports such that any two messages that are connected have opposite polarities. The fact that the attachments are injections but not necessarily bijections means that the correspondence may be partial: some of the messages of M_1 or M_2 may end up not being connected.

Proposition 2.10. *Every connection $\langle M_1 \overset{\mu_1}{\leftarrow} M \overset{\mu_2}{\rightarrow} M_2, \Lambda\rangle$ defines an injection $\langle \mu_1, \mu_2\rangle$ from A_M to $A_{M_1} \cup A_{M_2}$ as follows: for every $m \in M$ and $\{i, j\} = \{1, 2\}$, if $\mu_i(m) \in M_i^-$ then $\langle \mu_1, \mu_2\rangle(m!) = \mu_i(m)!$ and $\langle \mu_1, \mu_2\rangle(m¿) = \mu_j(m)¿$.*

Definition 2.11 (Asynchronous relational net)**.** *An asynchronous relational net (ARN) α consists of:*

- *A simple finite graph $\langle P, C\rangle$ where P is a set of nodes and C is a set of edges. Note that each edge is an unordered pair $\{p, q\}$ of nodes.*
- *A labelling function that assigns a process $\langle \gamma_p, \Lambda_p\rangle$ to every node p and a connection $\langle \gamma_c, \Lambda_c\rangle$ to every edge c such that:*
  - *If $c = \{p, q\}$ then γ_c is a pair of attachments $\langle M_p \overset{\mu_p}{\leftarrow} M_c \overset{\mu_q}{\rightarrow} M_q\rangle$ for some $M_p \in \gamma_p$ and $M_q \in \gamma_q$.*
  - *If $\gamma_{\{p,q\}} = \langle M_p \overset{\mu_p}{\leftarrow} M_{\{p,q\}} \overset{\mu_q}{\rightarrow} M_q\rangle$ and $\gamma_{\{p,q'\}} = \langle M'_p \overset{\mu'_p}{\leftarrow} M_{\{p,q'\}} \overset{\mu'_{q'}}{\rightarrow} M'_{q'}\rangle$ with $q \neq q'$, then $M_p \neq M'_p$ — i.e., different channels cannot share ports.*

We also define the following sets and mappings:

- *$A_\alpha = \bigcup_{p \in P} p.A_{\gamma_p}$ is the language associated with α.*
- *For every $p \in P$, ι_p is the function that maps A_{γ_p} to A_α, which prefixes the actions of A_{γ_p} with p.*
- *For every $c \in C$, ι_c is the function that maps A_{M_c} to A_α, which, assuming that $c = \{p, q\}$, translates the actions of A_{M_c} through $\langle p.\_ \circ \mu_p, q.\_ \circ \mu_q\rangle$.*
- *$\Lambda_\alpha = \{\lambda \in (2^{A_\alpha})^\omega \colon \forall p \in P(\lambda|_{\iota_p} \in \Lambda_p) \wedge \forall c \in C(\lambda|_{\iota_c} \in \Lambda_c)\}$.*

Note that, for every $p \in P$, $(\_|_{\iota_p})$ first removes the actions that are not in the language $p.A_p$ and then removes the prefix p. Similarly, for every $c = \{p, q\} \in C$, $(\_|_{\iota_c})$ first removes the actions that are not in the language $\langle p.\_ \circ \mu_p, q.\_ \circ \mu_q\rangle(A_{M_c})$, then removes the prefixes p and q, and then projects onto the language of M_c.

We often refer to the ARN through the quadruple $\langle P, C, \gamma, \Lambda\rangle$ where γ returns the set of ports of the processes that label the nodes and the pair of attachments of the connections that label the edges, and Λ returns the corresponding properties. The fact that the graph is simple – undirected, without self-loops or multiple edges – means that all interactions between two given processes are supported by a single channel and that no process can interact with itself. The graph is undirected because, as already mentioned, channels are bidirectional.

The alphabet of A_α is the union of the alphabets of the processes involved translated by prefixing all actions with the node from which they originate (see the definition of this translation after Def. 2.3).

We take the set Λ_α to define the set of possible traces observed on α – those traces over the alphabet of the ARN that are projected to traces of all its processes and channels. Notice that

$$\Lambda_\alpha = \bigcap_{p \in P} \iota_p(\Lambda_p) \cap \bigcap_{c \in C} \iota_c(\Lambda_c)$$

That is, the behaviour of the ARN is given by the intersection of the behaviour of the processes and channels translated to the language of the ARN — this corresponds to what one normally understands as a parallel composition.

Notice that nodes and edges denote *instances* of processes and channels, respectively. Different nodes (resp. edges) can be labelled with the same process (resp. channel), i.e., processes and channels act as *types*. This is why it is essential that, in the ARN, it is possible to trace actions to the instances of processes where they originate (all the actions of channels are mapped to actions of processes through the attachments so it is enough to label actions with nodes). Also notice that the alphabets of the channels are translated through the attachments in a way that is consistent with the translations performed on process alphabets.

In order to illustrate the notions introduced in the paper, we consider a simplified bank portal that mediates the interactions between clients and the bank in the context of different business operations such as the request for a credit. Fig. 2 depicts an ARN with two interconnected processes that implement that business operation. Process $Clerk$ is responsible for the interaction with the environment and for making decisions on credit requests, for which it relies on the process $RiskEvaluator$ that is able to evaluate the risk of the transaction.

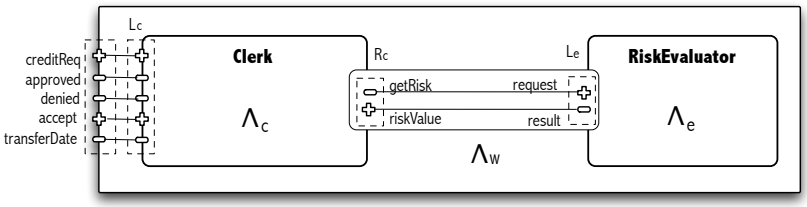

Figure 2: An example of an ARN with two processes connected through a channel.

The graph of this ARN consists of two nodes $c{:}Clerk$ and $e{:}RiskEvaluator$ and an edge $\{c, e\}{:}w_{ce}$ where:

- $Clerk$ is a process with two ports: L_c and R_c. In port L_c, the process receives messages $creditReq$ and $accept$ and sends $approved$, $denied$ and $transferDate$. Port R_c has outgoing message $getRisk$ and incoming message $riskValue$. The behaviour of $Clerk$ is as follows: immediately after the delivery of the first $creditReq$ message on port L_c, it publishes $getRisk$ on R_c; then it waits five steps for the delivery of $riskValue$, upon which it either publishes $denied$ or $approved$ (we abstract from the criteria that it uses for deciding on the credit); if $riskValue$ does not arrive by the deadline, $Clerk$ publishes $denied$ on L_c; after sending $approved$ (if ever), $Clerk$ waits twenty steps for the delivery of $accept$, upon which it sends $transferDate$; all other deliveries of $creditReq$ and $accept$ are discarded. The property that corresponds to this behaviour is denoted by Λ_c in Fig. 2.

- $RiskEvaluator$ is a process with a single port (L_e) with incoming message $request$ and outgoing message $result$. Its behaviour is quite simple: every time $request$ is delivered, it takes no more than three steps to publish $result$. The property that corresponds to this behaviour is denoted by Λ_e in Fig. 2.

- The port R_c of $Clerk$ is connected with the port L_e of $RiskEvaluator$ through $w_{ce}{:}\langle R_c \overset{\mu_c}{\leftarrow} \{m, n\} \overset{\mu_e}{\rightarrow} L_e, \Lambda_w\rangle$, with $\mu_c{=}\{m \mapsto getRisk, n \mapsto riskValue\}$,

$\mu_e = \{m \mapsto request, n \mapsto result\}$. The corresponding channel is reliable: a $m_¡$ follows every $m!$ and a $n_¡$ follows every $n!$. The property that corresponds to this behaviour is denoted by Λ_w in Fig. 2.

The alphabet of this ARN is the language of actions generated by the set M of messages defined by:

M^+**:** $c.creditReq$, $c.accept$, $c.riskValue$, $e.request$

M^-**:** $c.approved$, $c.denied$, $c.transferDate$, $c.getRisk$, $e.result$

In order to explain how the channels connect processes, consider a trace of the channel w_{ce} of the form:

$$\emptyset^{k_1} \cdot m! \cdot m_¡ \cdot \emptyset^{k_2} \cdot n! \cdot n_¡ \cdot \emptyset^{\omega}$$

The translation of this trace to the language of the ARN is a set of traces of the form

$$\tau_1 \cdot A_1 \oplus c.getRisk! \cdot A_2 \oplus e.request_¡ \cdot \tau_2 \cdot A_3 \oplus e.result! \cdot A_4 \oplus c.riskValue_¡ \cdot \lambda$$

where the τ_i are segments of length k_i, λ is a trace and A_i are sets of actions, none of which intersects $\{c.getRisk!, e.request_¡, e.result!, c.riskValue_¡\}$. We use $A \oplus a$ as shorthand for $A \cup \{a\}$.

That is, *Clerk* publishes $getRisk$, which the channel delivers to $RiskEvaluator$ as $request$; after a while,$RiskEvaluator$ publishes $result$, which the channel delivers to *Clerk* as $riskValue$, both without any delay.

A class of very simple ARNs, which we call *atomic*, are those that consist of a single process:

Definition 2.12 (ARN defined by a process)**.** *Given a process P, we define the ARN ν_P whose graph consists of only one node, which is labelled with P.*

2.3. Consistency

In [18], joint consistency of the descriptions of the processes and the connections is required for a (component-based) relational net to be well defined. For asynchronous relational nets, consistency can be formulated as follows:

Definition 2.13 (Consistent ARN)**.** *An ARN α is said to be* consistent *if Λ_α is not empty.*

Consistency of an ARN is an important property because it shows that there is a joint trace of all the processes and channels that are part of the ARN. Naturally, one cannot expect every ARN to be consistent as the interference established through the connections may make it impossible for the processes involved to work together. Therefore, it is important to know how one can determine whether an ARN is consistent. Typically, this could be done at the level of automata (e.g., non-deterministic Büchi automata [54]) that implement the processes and channels of the ARN by computing their product and checking that the resulting language is not empty, for example as in [53, 54]. However, for reasons explained in more detail in Sect. 2.4, we would like to have a more compositional way of checking for consistency, i.e., based on characteristics of the processes and channels involved without having to compute the product of the automata.

For that purpose, a different (but related) property was found to be relevant. Consistency is about infinite behaviours, i.e., it concerns the ability of all the processes and channels of an ARN to generate a full joint trace. However, it does not guarantee that, having engaged in a joint segment, the processes can proceed: it may happen that the joint segment is not a prefix of a joint (full) trace, which would be considered undesirable as it is not possible for individual processes to anticipate what other processes will do. Therefore, another intuitive (and important) property of an ARN is that, after any joint segment, a joint step can be performed[1].

Definition 2.14 (Progress-enabled ARN)**.** *For every ARN α, let*

$$\Pi_\alpha = \{\pi{\in}2^{A_\alpha}{}^*\colon \forall p{\in}P(\pi|_{\iota_p}{\in}\Lambda_p^f) \wedge \forall c{\in}C(\pi|_{\iota_c}{\in}\Lambda_c^f)\}$$

We say that α is progress-enabled iff

$$\forall \pi{\in}\Pi_\alpha.\exists A{\subseteq}A_\alpha(\pi{\cdot}A){\in}\Pi_\alpha.$$

The set Π_α consists of all the partial traces that the processes and channels can jointly engage in. Note that, because processes and channels are consistent, Π_α contains at least the empty segment. Because the intersection of A with the alphabet of any process or channel can be empty, being progress-enabled does not require all parties to actually perform an action. The set A itself can be empty and, indeed, the ARN can 'stutter' forever if its processes and channels have no requirements to perform actions.

By itself, being progress-enabled does not guarantee that an ARN is consistent: moving from finite to infinite behaviours requires the analysis of what happens 'at the limit'. A progress-enabled but inconsistent ARN guarantees that all the processes will happily make joint progress but at least one will be prevented from achieving a successful full trace at the limit. For example, consider the following two processes: P recurrently sends a given message m and Q is able to receive a message n but only a finite, though arbitrary, number of times. If these processes are interconnected through a reliable channel that ensures the delivery of n every time m is published, it is easy to conclude that the resulting ARN is not consistent in spite of being progress-enabled: after having engaged in any joint partial trace, both processes and the channel can proceed (Q will let the channel deliver n one more time if necessary); however, they are not able to generate a full joint trace because P will want to send m an infinite number of times and Q will not allow the channel to deliver n infinitely often.

A class of progress-enabled ARNs for which we can guarantee consistency are those that involve only closed (safety) properties (cf. Def. 2.2). The rationale is that, by choosing to work with safety properties, 'success' does not need to be measured at the limit: checking the ability to make 'good' progress is enough.

From a methodological point of view, considering ARNs that consist of safety properties is justified by the fact that, within SOC, we are interested in processes whose liveness properties are bounded (bounded liveness being itself a safety property). This is because, in typical business applications, one is interested only in services that respond within a fixed (probably negotiated) delay. In SOC, one does not offer as a service the kind of systems that, like operating systems, are not meant to terminate[2].

[1] Progress properties of this kind have been studied, for example, for communicating finite-state machines [13, 38].

[2] Cloud computing does offer platform or infrastructure as a service, but this is not what is normally meant by SOC — software as a service.

Definition 2.15 (Safe processes, channels and ARNs)**.** *A process $\langle \gamma, \Lambda \rangle$ (resp. channel $\langle M, \Lambda \rangle$) is said to be* safe *if Λ is closed. A* safe ARN *is one that is labelled with safe processes and channels.*

Proposition 2.16. *For every safe ARN α, Λ_α is a closed (safety) property.*

Proof. Λ_α is the intersection of the images of the properties of the processes and channels associated with the nodes and edges of the graph. According to Prop. 2.4, those images are safety properties. The result follows from the fact that an intersection of closed sets in any topology is itself a closed set. □

We can now prove one of the major results of this paper.

Theorem 2.17 (Consistency)**.** *Any safe progress-enabled ARN is consistent.*

Proof. Given that the processes and channels in a safe ARN are consistent, Π_α is not empty (it contains at least the empty segment ϵ). Π_α can be organised as a tree, which is finitely branching because A_α is finite. If the ARN is progress-enabled, the tree is infinite. By Kőnigs lemma, it contains an infinite branch λ.

We now prove that $\lambda \in \Lambda_\alpha$, i.e., $\lambda|_{\iota_p} \in \Lambda_p$ for all $p \in P$ and $\lambda|_{\iota_c} \in \Lambda_c$ for all $c \in C$:

1. Let $p \in P$ and $\pi \prec \lambda|_{\iota_p}$. We know that π is of the form $\pi'|_{\iota_p}$ where $\pi' \in \Pi_\alpha$. Therefore, $\pi \in \Lambda_p^f$.
2. It follows that $\lambda|_{\iota_p} \in \overline{\Lambda_p}$.
3. Because Λ_p is closed, we can conclude that $\lambda|_{\iota_p} \in \Lambda_p$.
4. The same reasoning applies to all channels.

□

It is not difficult to see that any atomic ARN ν_P, where P is a process such as $Clerk$, is progress-enabled. This is because the process is taken in isolation. For communicating finite-state machines, the problem of whether any arbitrary pair of machines is able to communicate indefinitely is known to be undecidable but approaches exist to bypass this problem [13, 37, 38]. In the next subsection, we analyse the composition of ARNs and give sufficient conditions for the composition of progress-enabled ARNs to be progress-enabled, which effectively guarantees that ARNs are progress-enabled by construction.

2.4. Composing ARNs

We consider now the composition operation of our component algebra. Two ARNs can be composed through the ports that are still available for establishing further interconnections, i.e., not connected to any other port, which we call interaction-points:

Definition 2.18 (Interaction-point)**.** *An interaction-point of an ARN $\alpha = \langle P, C, \gamma, \Lambda \rangle$ is a pair $\langle p, M \rangle$ such that $p \in P$, $M \in \gamma_p$ and there is no edge $\{p, q\} \in C$ labelled with a connection that involves M. We denote by I_α the collection of interaction-points of α.*

For example, the ARN depicted in Fig. 2 has a single interaction point, which is represented by projecting the corresponding port to the external box.

Interaction-points are used in the operation of composition that we define for ARNs, which subsumes the notion of interconnect of [18]:

Proposition and Definition 2.19 (Composition of ARNs)**.** *Let $\alpha_1 = \langle P_1, C_1, \gamma_1, \Lambda_1\rangle$ and $\alpha_2 = \langle P_2, C_2, \gamma_2, \Lambda_2\rangle$ be ARNs such that P_1 and P_2 are disjoint, and a family $w^i = \langle M_1^i \xleftarrow{\mu_1^i} M^i \xrightarrow{\mu_2^i} M_2^i, \Lambda^i\rangle$ $(i = 1 \ldots n)$ of connections for interaction-points $\langle p_1^i, M_1^i\rangle$ of α_1 and $\langle p_2^i, M_2^i\rangle$ of α_2 such that, for every $i \neq j$:*

- *$p_1^i \neq p_1^j$ or $p_2^i \neq p_2^j$,*
- *if $p_1^i = p_1^j$ then $M_1^i \neq M_1^j$,*
- *if $p_2^i = p_2^j$ then $M_2^i \neq M_2^j$.*

The composition $(\alpha_1 \;||^{i=1\ldots n}_{\langle p_1^i, M_1^i\rangle, w^i, \langle p_2^i, M_2^i\rangle}\; \alpha_2)$ is the ARN defined as follows:

- *Its graph is $\langle P_1 \cup P_2, C_1 \cup C_2 \cup \bigcup_{i=1\ldots n}\{p_1^i, p_2^i\}\rangle$*
- *Its labelling function coincides with that of α_1 and α_2 on the corresponding subgraphs, and assigns to the new edges $\{p_1^i, p_2^i\}$ the label w^i.*

Proof. We need to prove that the composition does define an ARN. This is because we are adding to the sum of the graphs edges between interaction-points that do not share interaction-points, so the resulting graph is simple. It is easy to check that the labels are well defined. □

Fig. 2 can also be used to illustrate the composition of ARNs: the depicted ARN is the composition of the two atomic ARNs defined by $Clerk$ and $RiskEvaluator$.

An important property of ARN composition is:

Proposition 2.20. *Let α be a composition of two ARNs as in Def. 2.19. Let ι_1 be the inclusion of A_{α_1} in A_α, ι_2 the inclusion of A_{α_2} in A_α, and ι_{c^i} the inclusion of $A_{M_{c^i}}$ in A_α where c^i is the channel involved in w^i. Then,*

$$\Lambda_\alpha = \iota_1(\Lambda_{\alpha_1}) \cap \iota_2(\Lambda_{\alpha_2}) \cap \bigcap_{i=1\ldots n} \iota_{c^i}(\Lambda^i)$$

Proof. The results follows from the definition of Λ_α given in Def. 2.11. □

We consider now the question of ascertaining that the composition of two consistent ARNs is consistent. In SOC, composition of ARNs occurs at run time as applications discover and bind to applications that offer required services. Therefore, checking for consistency by checking directly that the set of traces generated by the composition is not empty (which could be done, as already mentioned, over the product of the automata that implement the processes and channels) is not feasible. Instead, we would like to find criteria over the ARNs and the channels that guarantee consistency of the composition and can be checked at design time.

We start by identifying criteria for the composition of two progress-enabled ARNs to be progress-enabled. For this purpose, an important property of an ARN relative to its set of interaction-points is that it does not constrain the actions that do not 'belong' to the ARN. Naturally, this needs to be understood in terms of a computational and communication model in which it is clear what dependencies exist between the different parties. Taking it to be the responsibility of processes to publish and process messages, and of channels to deliver them, we are interested in processes that are able to buffer incoming messages, i.e., are 'delivery-enabled', and channels that are able

to buffer published messages, i.e., are 'publication-enabled'. Note that processes are nevertheless free to discard delivered messages and channels not to deliver published messages.

Definition 2.21 (Delivery-enabled ARN)**.** *Let $\alpha{=}\langle P, C, \gamma, \Lambda\rangle$ be an ARN, $\langle p, M\rangle{\in}I_\alpha$ one of its interaction-points, and $D_{\langle p,M\rangle}{=}\{p.m¡ : m{\in}M^+\}$. We say that α is* delivery-enabled *in relation to $\langle p, M\rangle$ if, for every $(\pi{\cdot}A){\in}\Pi_\alpha$ and $B{\subseteq}D_{\langle p,M\rangle}$, we have that $(\pi{\cdot}B \cup (A{\setminus}D_{\langle p,M\rangle})){\in}\Pi_\alpha$.*

That is, being delivery-enabled at an interaction point requires that any joint prefix of the ARN (see Def. 2.14 for Π_α) can be extended by any set of messages delivered at that interaction-point. Notice that this does not interfere with the decision of the process to publish messages: $B{\cup}(A{\setminus}D_{\langle p,M\rangle}))$ retains all the publications present in A.

Consider, for example, a process P with a single port with an incoming message n and an outgoing message m. If the process P ensures the publication of m immediately after the delivery of the first n and does not publish m in any other situation, then its set of behaviors can be expressed as

$$\emptyset^* \cdot (\{n¡\} \cdot \{m!\} + \{n¡\} \cdot \{m!, n¡\} + \emptyset) \cdot (\emptyset + \{n¡\})^\omega$$

It is easy to see that the atomic ARN defined by P is delivery-enabled in relation to its interaction-point.

Definition 2.22 (Publication-enabled channel)**.** *Let $h{=}\langle M,\Lambda\rangle$ be a channel and $E_h = \{m!{:}m{\in}M\}$. We say that h is* publication-enabled *iff, for every $(\pi{\cdot}A){\in}\Lambda^f$ and $B{\subseteq}E_h$, we have that $\pi{\cdot}(B{\cup}(A{\setminus}E_h)){\in}\Lambda^f$.*

The requirement here is that any prefix can be extended by the publication of any set of messages, i.e., the channel should not prevent processes from publishing messages when they are in a state in which they could do so. Notice that this does not interfere with the decision of the channel to deliver messages: $(B{\cup}(A{\setminus}E_h))$ retains all the deliveries present in A.

Consider, for example, a channel with a singleton set of messages $\{n\}$. If the channel ensures the delivery of n immediately after its publication in the channel, then its set of behaviors corresponds to the language of the Bücchi automaton presented in Fig. 3. It is easy to see that this channel is publication-enabled.

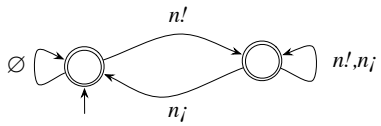

Figure 3: The behaviour of a publication-enabled channel.

We can now prove our main composition result:

Theorem 2.23. *Let α be a composition of progress-enabled ARNs through the connections $w^i = \langle M_1^i \overset{\mu_1^i}{\leftarrow} M^i \overset{\mu_2^i}{\rightarrow} M_2^i, \Lambda^i\rangle$, $i = 1 \ldots n$, i.e.,*

$$\alpha = (\alpha_1 \;||\!|^{i=1\ldots n}_{\langle p_1^i, M_1^i\rangle, w^i, \langle p_2^i, M_2^i\rangle}\; \alpha_2)$$

If, for each i=1...n, α_1 is delivery-enabled in relation to $\langle p_1^i, M_1^i\rangle$, α_2 is delivery-enabled in relation to $\langle p_2^i, M_2^i\rangle$ and h^i=$\langle M^i, \Lambda^i\rangle$ is publication-enabled, then α is progress-enabled.

Proof. To simplify the notation, we consider the case of a single pair of interaction points — $\alpha = (\alpha_1 \|_{\langle p, M_1\rangle, w, \langle q, M_2\rangle} \alpha_2)$ is a composition of progress-enabled ARNs where $w = \langle M_1 \overset{\mu_1}{\leftarrow} M \overset{\mu_2}{\rightarrow} M_2, \Lambda\rangle$, α_1 is delivery-enabled in relation to $\langle p, M_1\rangle$, α_2 is delivery-enabled in relation to $\langle q, M_2\rangle$ and $h = \langle M, \Lambda\rangle$ is publication-enabled. We prove that α is progress-enabled.

Let $c = \{p, q\}$, π∈Π_α and π_1, π_2 and π_c be the corresponding projections to the languages of α_1, α_2 and A_M, respectively. Let Π_c denote Λ^f, the set of prefixes of the traces of the channel h that labels the edge c.

We know that π_1∈Π_{α_1}, π_2∈Π_{α_2} and π_c∈Π_c. Because α_1 and α_2 are progress-enabled, let $(\pi_1{\cdot}B_1)$∈Π_{α_1} and $(\pi_2{\cdot}B_2)$∈Π_{α_2}. Let also $(\pi_c{\cdot}B_c)$∈Π_c. The addition of the new edge c only interferes with the ability of p and q to move — the language $\iota_c(A_M)$ only intersects those of the two interaction-points. Therefore, we need to adjust the deliveries in B_1 and B_2 with the intersection of $\mu_1(B_c)$ with $D_{\langle p, M_1\rangle}$ and of $\mu_2(B_c)$ with $D_{\langle q, M_2\rangle}$, respectively — the deliveries made by the channel — and B_c with the intersection of $\mu_1^{-1}(B_1) \cup \mu_2^{-1}(B_2)$ with E_h — the publications made into the channel.

Let B_p=$\mu_1(B_c)\cap D_{\langle p, M_1\rangle}$ and B_q=$\mu_2(B_c)\cap D_{\langle q, M_2\rangle}$:

- Let B_1'=$B_p \cup (B_1 \backslash D_{\langle p, M_1\rangle}))$. Then, $(\pi_1{\cdot}B_1')$∈Π_{α_1} because α_1 is delivery-enabled in relation to $\langle p, M_1\rangle$ — that is, α_1 progresses by the deliveries made by c and the publications made by B_1.
- Let B_2'=$B_q \cup (B_2 \backslash D_{\langle q, M_2\rangle}))$. Then, $(\pi_2{\cdot}B_2')$∈Π_{α_2} because α_2 is delivery-enabled in relation to $\langle q, M_2\rangle$ — that is, α_2 progresses by the deliveries made by c and the publications made by B_2.
- Let $B_c' = (\mu_1^{-1}(B_1) \cup \mu_2^{-1}(B_2)) \cap E_h$. Then, $(\pi_c{\cdot}(\mu_1^{-1}(B_p) \cup \mu_2^{-1}(B_q) \cup B_c'))$∈$\Pi_c$ because h is publication-enabled, i.e., the channel progresses by the deliveries made by B_c and the publications made by B_1 and B_2. Note that because μ_1 and μ_2 are injective, $(B_c \backslash E_h)$=$(\mu_1^{-1}(B_p) \cup \mu_2^{-1}(B_q))$.

We can now conclude that $(\pi{\cdot}(\iota_1(B_1') \cup \iota_2(B_2') \cup \iota_c(B_c')))$∈$\Pi_\alpha$. □

Because the composition of safe ARNs through safe channels is safe, Theorem 2.23 can be generalised to guarantee consistency of composition:

Corollary 2.24 (Consistency of composition)**.** *The composition of safe progress-enabled ARNs is both safe and progress-enabled (and, hence, consistent) provided that interconnections are made through safe publication-enabled channels and over interaction-points in relation to which the ARNs are delivery-enabled.*

Another useful result is that checking whether an ARN is delivery-enabled in relation to an interaction-point can be reduced to checking that the process that owns the interaction-point, as an atomic ARN, is delivery-enabled.

Proposition 2.25. *Let α be the composition of two ARNs α_1 and α_2 through the connections $w^i = \langle M_1^i \overset{\mu_1^i}{\leftarrow} M^i \overset{\mu_2^i}{\rightarrow} M_2^i, \Lambda^i\rangle$, $i = 1 \ldots n$.*

- *Let* $\langle p'_1, M'_1 \rangle$ *be an interaction-point of* α_1 *that is not one of the* $\langle p^i_1, M^i_1 \rangle$*. If* α_1 *is delivery-enabled in relation to* $\langle p'_1, M'_1 \rangle$*, so is* α*.*
- *Let* $\langle p'_2, M'_2 \rangle$ *be an interaction-point of* α_2 *that is not one of the* $\langle p^i_2, M^i_2 \rangle$*. If* α_2 *is delivery-enabled in relation to* $\langle p'_2, M'_2 \rangle$*, so is* α*.*

Therefore, the proof that an ARN is progress-enabled can be reduced to checking that individual processes are delivery-enabled in relation to their interaction points and that the channels used for composition are publication-enabled. That is, all the checking can be done at design time, not necessarily at composition time (which, in SOC, takes place at run time).

In order to consider the complexity of checking these properties, we need to consider implementation models for processes and channels. Typical examples of models are finite automata of some kind. For automata that enforce a syntactic distinction between input and output actions (i.e., between actions generated by the environment or by the automata), the notions of delivery/publication-enabled are subsumed by the property of being input-enabled, taking that publications are inputs for channels and deliveries are inputs for processes. For example, I/O automata [43] are, by definition, input-enabled.

Consider, however, the more general class of *non-deterministic Bücchi automata* (NBAs) [54]. An NBA over an alphabet A is a tuple of the form $\langle Q, \delta, Q_0, Q_\infty \rangle$ where Q is a finite set of states, $Q_0 \subseteq Q$ is the subset of initial states, $Q_\infty \subseteq Q$ is the set of accepting states, and $\delta : Q \times A \to 2^Q$ is the transition relation. The property defined by $\langle Q, \delta, Q_0, Q_\infty \rangle$ is the set of infinite sequences of elements of A that, starting in an initial state, generate a run that visits at least one of the accepting states infinitely often.

In relation to safety properties, there is also a *closure* operator on NBAs [3]: the closure of $\langle Q, \delta, Q_0, Q_\infty \rangle$ is $\langle Q, \delta, Q_0, Q \rangle$, i.e., the NBA obtained by making all states accepting. A reduced NBA (i.e., one of which every state leads to an accepting state) defines a safety property if and only if its closure defines the same property. Furthermore, every NBA is equivalent to a reduced one.

Therefore, given that we are interested in working with safe processes and channels, we can choose closed reduced NBAs as models of their implementations. In this case, it is easy to see that all that needs to be checked for processes (resp. channels) to be delivery (resp. publication) enabled is that, from every state of the automata that implement them, the set of transitions from that state satisfies the corresponding property, i.e., for every set of deliveries (resp. publications), there is a transition that delivers (resp. publishes) exactly those messages. As a result, the complexity of the checking process is in the order of $(|Q| \times m \times 2^{2 \times m})$ where m is the size of the largest port of the ARN.

From a methodological point of view, ensuring that processes are delivery-enabled comes 'naturally': on the one hand, it is a matter of ensuring that, no matter what state the process is in, deliveries are not refused – this can be achieved, for example, by letting each process have its own buffer for incoming messages; on the other hand, it is a matter of ensuring independence of outputs in relation to inputs received during the same step – for example, forbidding synchronisation between inputs into the delivery buffer and outputs (publication). Notice that processes can be programmed to discard certain inputs (deliveries) on given states – unless explicitly required, to do so, processes can discard deliveries.

2.5. Abstraction

One question that arises quite naturally is whether, by forgetting its internal structure, an ARN could be seen as a process – an abstraction mechanism useful in system design. For example, in the example of the composition of $Clerk$ and $RiskEvaluator$ (see Fig. 2), this would be a process that would have L_c as its only port and whose behaviour would be that of the ARN translated back to the language of L_c.

When considering an ARN as a process, its ports should be, intuitively, the interaction points, i.e., those ports of the processes that are still available for establishing further interconnections.

Definition 2.26 (Process defined by an ARN)**.** *Let α be a consistent ARN. We define the process P_α as follows:*

- *Its set of ports γ_α consists of the ports $p.M$ where $\langle p, M\rangle$ is an interaction point of α.*
- *Its behaviour consists of the projection of Λ_α onto the language of A_{γ_α}, i.e., $\Lambda_\alpha|_{A_{\gamma_\alpha}}$.*

Notice that α being consistent, $\Lambda_\alpha|_{A_{\gamma_\alpha}}$ is not empty and, therefore, P_α is indeed a process. Also note that we need to apply the translations $p.\_$ to the ports of the interaction points to ensure that γ_α consists of mutually-disjoint points as required in Def. 2.7. In practice, we can omit these translations if the original ports are disjoint.

We consider now a relationship of abstraction between two ARNs. First, we consider abstraction between processes:

Definition 2.27 (Process morphism)**.** *A morphism between two processes $\langle \gamma_1, \Lambda_1\rangle$ and $\langle \gamma_2, \Lambda_2\rangle$ consists of*

- *A function $\sigma : \gamma_1 \to \gamma_2$.*
- *For every $M{\in}\gamma_1$, a polarity-preserving function $\sigma_M{:}M{\to}\sigma(M)$, i.e., σ_M maps M^+ to $\sigma(M)^+$ and M^- to $\sigma(M)^-$.*

such that $\Lambda_2|_{\sigma^*}{\subseteq}\Lambda_1$ where by σ^* we denote the translation between A_{γ_1} and A_{γ_2} defined by: for every $M{\in}\gamma_1$ and $m{\in}M$, $\sigma^*(m!){=}\sigma_M(m)!$ and mutatis mutandis for $m¡$.

Two processes are said to be isomorphic *if there is a morphism between them that is bijective and whose inverse is also a morphism.*

For simplicity, we use σ to denote σ^* and also the morphism $\langle \sigma, \{\sigma_M : M{\in}\gamma_1\}\rangle$.

Isomorphic processes are equal up to a renaming of their alphabets. An example is, for every process Q, the process P_{ν_Q} defined by the atomic ARN that consists of Q.

To capture abstraction, we are particularly interested in the process morphisms where the map between the set of ports is injective and each map between two ports is also injective — in the abstract process we can forget ports and we can forget messages within ports of the concrete process, but we cannot split ports or duplicate messages of the concrete process. We call such morphisms *injective*.

Process morphisms can be generalised to a relationship of abstraction between ARNs as follows:

Definition 2.28 (Abstraction for ARNs)**.** *An abstraction of an ARN β consists of*

- *an ARN α,*

- *an injective morphism ρ:$P_\alpha \to P_\beta$ between the processes defined by the two ARNs, i.e., $\Lambda_\beta|_\rho \subseteq \Lambda_\alpha$.*

That is, the abstraction may remove some interaction-points of β and some of the messages of those interaction-points, and it preserves the behaviour of β over the remaining alphabet of messages. In other words, abstraction is a relationship between the behaviours that can be observed at the interaction-points of the two ARNs.

We write $\beta \preceq_\rho \alpha$ to indicate that α is an abstraction of β, or just $\beta \preceq \alpha$ when we do not want to refer to the abstraction morphism. The abstraction relation thus defined is reflexive (the identity is an abstraction) and transitive (abstractions compose).

This notion of abstraction is compositional in the following sense:

Theorem 2.29 (Compositionality of abstraction)**.** *Given a composition*

$$\beta = (\beta_1 \;||_{\langle p_1, M_1\rangle, w, \langle p_2, M_2\rangle}\; \beta_2)$$

with $w = \langle M_1 \overset{\mu_1}{\leftarrow} M \overset{\mu_2}{\rightarrow} M_2, \Lambda\rangle$, and

$$\alpha = (\alpha_1 \;||_{\langle p'_1, M'_1\rangle, w', \langle p_2, M_2\rangle}\; \beta_2)$$

where $\beta_1 \preceq_\rho \alpha_1$, $\langle p_1, M_1\rangle = \rho(\langle p'_1, M'_1\rangle)$ and $w' = \langle M_1 \overset{\rho^{-1}\circ\mu_1}{\longleftarrow} \mu_1^{-1}(\rho(M'_1)) \overset{\mu_2}{\rightarrow} M_2, \Lambda\rangle$, then

$$(\beta \preceq_{\rho'} \alpha)$$

where ρ' coincides with ρ on $I_\alpha \backslash \langle p'_1, M'_1\rangle$ and with the identity on $I_{\beta_2} \backslash \langle p_2, M_2\rangle$.

Proof. See Fig. 4 for the context of the proof. We start by noticing that ρ' is well defined and injective. The result follows from Prop. 2.20 and the fact that, because $\beta_1 \preceq_\rho \alpha_1$, $\Lambda_{\beta_1}|_\rho \subseteq \Lambda_{\alpha_1}$. □

That is, in a composition of two ARNs, if we replace a component by one of its abstractions, the resulting composition is an abstraction of the original one. Notice that, because ρ is not necessarily surjective, w' may make fewer connections than w. The definition can be easily generalised to compositions via multiple connections.

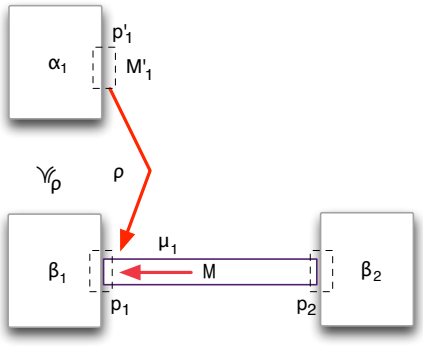

Figure 4: The context of Theorem 2.29.

Another important construction is the one through which we can observe the behaviour of an ARN through one of its interaction points:

Definition 2.30 (Process defined by an interaction point)**.** *Let α be a consistent ARN and $\langle p, M\rangle$ one of its interaction points. We define the process $P_{\alpha,p,M}$ as follows:*

- *Its only port is M.*
- *Its behaviour is $\Lambda_\alpha|_{\iota_{\alpha,p,M}}$ where $\iota_{\alpha,p,M}$ is the inclusion of A_M in A_α, i.e., the projection of Λ_α onto the language of A_M.*

That is, we project Λ_α onto the language of $p.A_{\gamma_p}$, then we remove the prefix p, and then the actions not in A_M.

Proposition 2.31. *Let $\alpha_1 = \langle P_1, C_1, \gamma_1, \Lambda_1\rangle$ and $\alpha_2 = \langle P_2, C_2, \gamma_2, \Lambda_2\rangle$ be two consistent ARNs such that P_1 and P_2 are disjoint. Let $w = \langle M_1 \overset{\mu_1}{\leftarrow} M \overset{\mu_2}{\rightarrow} M_2, \Lambda\rangle$ be a connection for interaction-points $\langle p_1, M_1\rangle$ of α_1 and $\langle p_2, M_2\rangle$ of α_2. Let*

$$\alpha = (\alpha_1 \Big|\Big|_{\langle p_1,M_1\rangle,w,\langle p_2,M_2\rangle} \alpha_2) \quad \textit{and} \quad \beta = (\alpha_1 \Big|\Big|_{\langle p_1,M_1\rangle,w,\langle p_2,M_2\rangle} P_{\alpha_2,p_2,M_2})$$

i.e., β is the composition of α_1 with the atomic ARN that consists of the single node p_2 labelled with the process P_{α_2,p_2,M_2} through the same connection w — i.e., we replace α_2 in α with the process that it defines for $\langle p_2, M_2\rangle$.

Then,

1. *$\alpha \preceq \beta$*
2. *Given any interaction-point $\langle p, M\rangle$ of α_1 such that $p_1 \neq p$, $P_{\alpha,p,M} = P_{\beta,p,M}$ — i.e., $\Lambda_\alpha|_{\iota_{\alpha,p,M}} = \Lambda_\beta|_{\iota_{\beta,p,M}}$*

Proof. Tedious. □

That is, in a composition of two ARNs, if we replace a component by the process that it defines at the interconnection point, the resulting composition is an abstraction of the original one. The second property further tells us that, for the purpose of observing the behaviour of α in relation to an interaction-point of α_1 other than the one used in the interconnection, only the behaviour that α_2 displays at the interconnection point is relevant.

Proposition 2.32. *Let α be a consistent ARN and $\langle p, M\rangle$ one of its interaction points. If $\beta \preceq_\rho \alpha$ and $\rho(\langle p, M\rangle)$ is an interaction point of β, then ρ_M is a process morphism $P_{\alpha,p,M} \rightarrow P_{\beta,\rho(\langle p,M\rangle)}$.*

3. The interface algebra

In this section, we put forward a notion of interface for software components described in terms of ARNs and a notion of interface composition that is suitable for service-oriented design. As discussed in Section 1, this means that interfaces need to specify the services that customers can expect from ARNs as well as the dependencies that the ARNs may have on external services for providing the services that they offer. Therefore, our notion of service interface consists of a provides-point (offering properties to customers) and a collection of requires-points, each of which specifies the properties of an external service that may be required and of the channel through which the external service will be connected. At the level of interfaces, properties (offered and required) are specified through logical formulas: in Section 3.1, we define the general properties that are required of a specification logic, of which we then give an example in Section 3.5.

We then define, in Section 3.2, what it means for an ARN to orchestrate an interface and discuss the problem of ensuring that orchestrations are well defined, i.e., that when interconnected with orchestrations of required services, the resulting composition is consistent and delivers the properties offered at the provides-point.

Two operations over interfaces are defined: composition in Section 3.3, and refinement in Section 3.4. Composition takes two interfaces and a match between a requires-point of one of the interfaces (the client) and the provides-point of the other (the supplier) – the properties provided by the supplier entail those required by the client. Refinement strengthens the properties offered at the provides-point and weakens those of the requires-points. Refinement is then shown to be compositional in relation to the composition operation. Finally, we show that orchestrations are compositional in relation to both the composition and refinement of interfaces.

3.1. Specifications

In [18], predicates are used as a means of describing properties of input/output behaviour, i.e., establishing relations (or the lack thereof) between inputs and outputs of processes, leading to several classes of relational nets depending on when they are considered to be 'well-formed'. In the context of our asynchronous communication model, behaviour is observed in terms of the actions that are performed over the lifetime of the process, for which the natural formalism to use is temporal logic, namely some form of linear temporal logic given that behaviours are defined in terms of infinite sequences of actions.

Rather than propose one specific logic, it seems to be more useful to identify the properties that would make a logic suitable for the description of the kind of networks with which we are concerned. Some of these properties concern the ability to relate specifications over alphabets related by a map, namely to relate the logical properties of processes with those of the networks in which they execute.

Definition 3.1 (Specification logic)**.** *A specification logic maps:*

- *every alphabet A to a set Ω_A (of sentences) and a (satisfaction) relation $\models_A$ between traces and sentences*
  - *Given a property Λ, we write $\Lambda \models_A \phi$ to mean that $\lambda \models_A \phi$ for every $\lambda \in \Lambda$, in which case we say that Λ is a model of (or validates) ϕ*
  - *Given $\Phi \subseteq \Omega_A$, we write $\Lambda \models_A \Phi$ to mean that $\Lambda \models_A \phi$ for every $\phi \in \Phi$*
  - *Given $\Phi \subseteq \Omega_A$, we denote by Λ_Φ the set of traces λ such that $\lambda \models_A \Phi$*
- *every function $\sigma : A \to B$ between alphabets to a function $\Omega_\sigma : \Omega_A \to \Omega_B$ that translates sentences over A to sentences over B such that, for every trace λ_B in B and sentence $\phi_A \in \Omega_A$,*

$$(1) \quad \lambda_B \models_B \Omega_\sigma(\phi_A) \text{ iff } \lambda_B|_\sigma \models_A \phi_A$$

We call a specification *a pair $\langle A, \Phi \rangle$ where A is an alphabet and Φ is a finite set of sentences in Ω_A.*

For simplicity, we tend to use $\sigma(\phi)$ as an abbreviation of $\Omega_\sigma(\phi)$.

Equation (1) means that the satisfaction relation is independent of the alphabet, i.e., that the specific choice of actions does not interfere with the logical properties of the satisfaction relation.

Proposition 3.2 (Algebraic properties of specification logics)**.** *Specification logics define institutions [36]. The following definitions and properties apply to all institutions. Let A be an alphabet and $\sigma{:}A{\to}B$ a map.*

1. $\models_A$ *extends to sets of sentences Φ over A as follows: $\Phi \models_A \phi$ iff, for every trace λ over A, if $\lambda \models_A \phi'$ for all $\phi' \in \Phi$, then $\lambda \models_A \phi$.*
2. $\Lambda \models_A \Phi$ *iff* $\Lambda \subseteq \Lambda_\Phi$
3. *For every set Φ of sentences over A and sentence ϕ, if $\Phi \models_A \phi$ then $\sigma(\Phi) \models_B \sigma(\phi)$.*

Definition 3.3 (Process defined by a specification)**.** *Let γ be a set of mutually disjoint ports and Φ a consistent set of sentences over A_γ. The process defined by the specification $\langle A_\gamma, \Phi\rangle$ is $\langle \gamma, \Lambda_\Phi\rangle$.*

Definition 3.4 (Processes and ARNs as models)**.** *The process $\langle \gamma, \Lambda\rangle$ is a model of (or validates) the specification $\langle A, \Phi\rangle$ via the alphabet map $\sigma{:}A{\to}A_\gamma$, which we denote by $\langle \gamma, \Lambda\rangle \models_\sigma \Phi$, iff $\Lambda|_\sigma \models \Phi$ — i.e., σ is a process morphism $\langle \gamma, \Lambda_\Phi\rangle \to \langle \gamma, \Lambda\rangle$.*

An ARN α is a model of (or validates) the specification $\langle A, \Phi\rangle$ via the alphabet map $\sigma{:}A{\to}A_{\gamma_\alpha}$, which we denote by $\alpha \models_\sigma \Phi$, iff $P_\alpha \models_\sigma \Phi$.

Proposition 3.5. *The following properties follow immediately from the definition:*

- *If $\rho{:}P_1{\to}P_2$ is a process morphism and $P_1 \models_\sigma \Phi$ then $P_2 \models_{\rho\circ\sigma} \Phi$.*
- *If $\beta \preceq_\rho \alpha$ and $\alpha \models_\sigma \Phi$ then $\beta \models_{\rho\circ\sigma} \Phi$.*

We defer the discussion of choosing a specification logic to Sect. 3.5 and, for the rest of this section, we assume a fixed specification logic. Naturally, it would be important to show that at least one such logic exists. A simple example is linear temporal logic (LTL) [44]: a proof that LTL is an institution can be found in [24]. However, as discussed below, LTL is not necessarily the most suitable logic for defining an interface theory for safe ARNs.

3.2. Service interfaces and their orchestrations

In our model, a service interface identifies a number of ports through which services are provided and ports through which services are required (hence the importance of ports for correlating messages that belong together from a business point of view). Sentences of the specification logic are used for specifying the properties offered or required.

Ports for required services include messages as sent or received by the external service. Therefore, to complete the interface we need to be able to express requirements on the channel through which communication with the external service will take place, if and when required. In order to express those properties, we need to have actions on both sides of the channel, for which we introduce the notion of dual port.

Definition 3.6 (Dual port)**.** *Given a port M, we denote by M^{op} the port defined by $M^{op+} = M^-$ and $M^{op-} = M^+$.*

Definition 3.7 (Service interface)**.** *A service interface i consists of:*

- *A set I (of interface-points) partitioned into a singleton set $\{i^{\to}\}$ and a set $I^{\leftarrow}$ the members of which are called the* provides- *and* requires-points*, respectively.*
- *For every $r \in I$,*

– *a port* M_r*,*

– *a consistent set of formulas* Φ_r *over* A_{M_r}*.*

- *For every point* $r{\in}I^{\leftarrow}$*, a consistent set of formulas* Ψ_r *over* $A_{M_r} \cup A_{M_r^{op}}$*.*

We identify an interface with the tuple $\langle i^{\rightarrow}, I^{\leftarrow}, M, \Phi, \Psi\rangle$ where $M_r{:}r{\in}I$, $\Phi_r{:}r{\in}I$, $\Psi_r{:}r{\in}I^{\leftarrow}$ are the indexed families that identify the ports and specifications of each point of the interface.

In Fig. 5, we present an example of an interface for a credit service using a graphical notation similar to that of SCA. On the left, we have a provides-point $Customer$ and, on the right, a requires-point $IRiskEvaluator$.

The set of sentences Φ_c specifies the service offered at $Customer$. In the logic defined in Sect. 3.5, these are:

- $(creditReq_{¡}\ \mathcal{R}\ (creditReq_{¡} \supset \Diamond_{\leq 10}(approved! \vee denied!)))$ — either $approved$ or $denied$ are published within ten steps of the first delivery of $creditReq$.
- $\Box(approved! \supset (accept_{¡}\ \mathcal{R}_{\leq 20}\ (accept_{¡} \supset \Diamond_{\leq 2} transferDate!)))$ — if $accept$ is received within twenty steps of the publication of $approved$, $transferDate$ will be published within two steps.

The specification $IRiskEvaluator$ requires the external service to react to the delivery of every $getRisk$ by publishing $riskValue$ in no more than four steps:

$$\Phi_r :\ \Box(getRisk_{¡} \supset \Diamond_{\leq 4} riskValue!)$$

The channel is specified to be reliable with delay 1:

$$\Psi_r :\ \Box(getRisk! \supset \bigcirc getRisk_{¡}) \wedge \Box(riskValue! \supset \bigcirc riskValue_{¡})$$

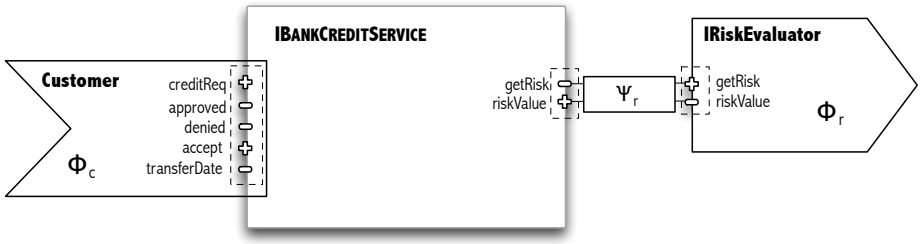

Figure 5: An example of a service interface.

An ARN orchestrates a service-interface by assigning interaction-points to interface-points in such a way that the behaviour of the ARN validates the specifications of the provides-points on the assumption that it is interconnected to ARNs that validate the specifications of the requires-points through channels that validate the corresponding specifications.

Definition 3.8 (Orchestration)**.** *An orchestration of a service interface* $\langle i^{\rightarrow}, I^{\leftarrow}, M, \Phi, \Psi\rangle$ *consists of:*

- *an ARN* $\alpha = \langle P, C, \gamma, \Lambda\rangle$ *where* P *and* I *are disjoint;*

- *an injective function* θ: $I \to I_\alpha$ *that assigns a different interaction-point to each interface-point; we write* $r \xrightarrow{\theta} p$ *to indicate that* $\theta(r) = \langle p, M_p \rangle$ *for some port* M_p*;*
- *a polarity-preserving injection* $\theta_{i\rightarrow}$ *from* $M_{i\rightarrow}$ *to* M_p *where* $i^{\rightarrow} \xrightarrow{\theta} p$*, which assigns to every message of the provides-point a message of the corresponding interaction-point of the ARN;*
- *for every requires-point* $r \in I^{\leftarrow}$*, a polarity-preserving injection* $\theta_r : M_r^{op} \to M_q$ *where* $r \xrightarrow{\theta} q$*, which assigns to every message of the requires-point a message of the corresponding interaction-point of the ARN — the polarities are reversed because the requires-point stands for an external service with which the ARN may be required to connect;*

such that $P_{\alpha^*,\theta(i\rightarrow)} \vDash_{\theta_{i\rightarrow}} \Phi_{i\rightarrow}$ *where:*

- *For every requires-point* r *of* $I^{\leftarrow}$*,* α_r *is the ARN defined by the process* $\langle \{M_r\}, \Lambda_{\Phi_r} \rangle$ *and* w_r *the connection* $\langle M_q \xleftarrow{\theta_r} M_r \xrightarrow{id} M_r, \Lambda_{\Psi_r} \rangle$ *where* $r \xrightarrow{\theta} q$*.*
- $\alpha^* = (\alpha \parallel^{r \in I^{\leftarrow}}_{\theta(r), w_r, \langle r, M_r \rangle} \alpha_r)$

That is, $\Lambda_{\alpha^*}|_{\iota_{\alpha^*,\theta(i\rightarrow)} \circ \theta_{i\rightarrow}} \vDash \Phi_{i\rightarrow}$ *where* $i^{\rightarrow} \xrightarrow{\theta} p$ *and* $\iota_{\alpha^*,\theta(i\rightarrow)}$ *is the inclusion of* A_{M_p} *in* A_{α^*}*.*

Notice that $P_{\alpha^*,\theta(i\rightarrow)}$ is the process that abstracts the behaviour of α^* observed at the image of the provides-point. The requirement is, therefore, that whenever the orchestration α is composed with 'canonical' implementations of the requires-points — the processes $\langle \{M_r\}, \Lambda_{\Phi_r} \rangle$, the resulting ARN validates the provides-point.

We borrow from [18] the notation $\alpha \lhd_\theta i$ to indicate that the ARN α provides, through the family of mappings θ, an orchestration of the service interface i. Figure 6 summarises the constructions involved in the definition.

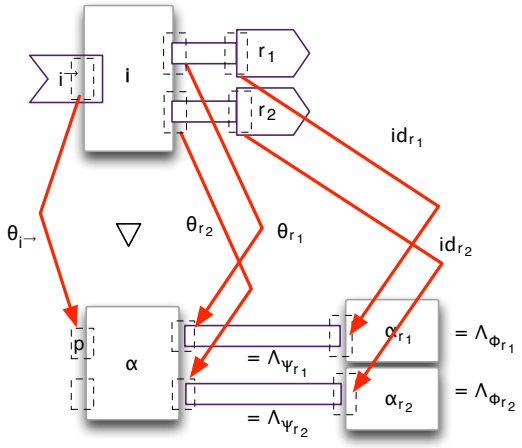

Figure 6: $\alpha \lhd_\theta i$ iff the properties in $\Phi_{i\rightarrow}$ are validated by the composition of α with the models of the specifications of the requires-points.

In order to illustrate the concept, consider again the atomic ARN ν_{Clerk}, defined by the process $Clerk$. As illustrated in Fig. 7, $\nu_{Clerk} \lhd_\theta$ IBANKCREDITSERVICE where IBANKCREDITSERVICE is the service interface presented before and θ is such that:

- $Customer \xrightarrow{\theta} \langle Clerk, L_c \rangle$ and $\theta_{Customer}$ is the identity function;
- $IRiskEvaluator \xrightarrow{\theta} \langle Clerk, R_c \rangle$ and $\theta_{IRiskEvaluator}$ is the identity function.

The property Λ_c is such that it validates Φ_c on the assumption that $Clerk$ is interconnected through $IRiskEvaluator$ to an ARN that validates Φ_r via a channel that validates Ψ_r.

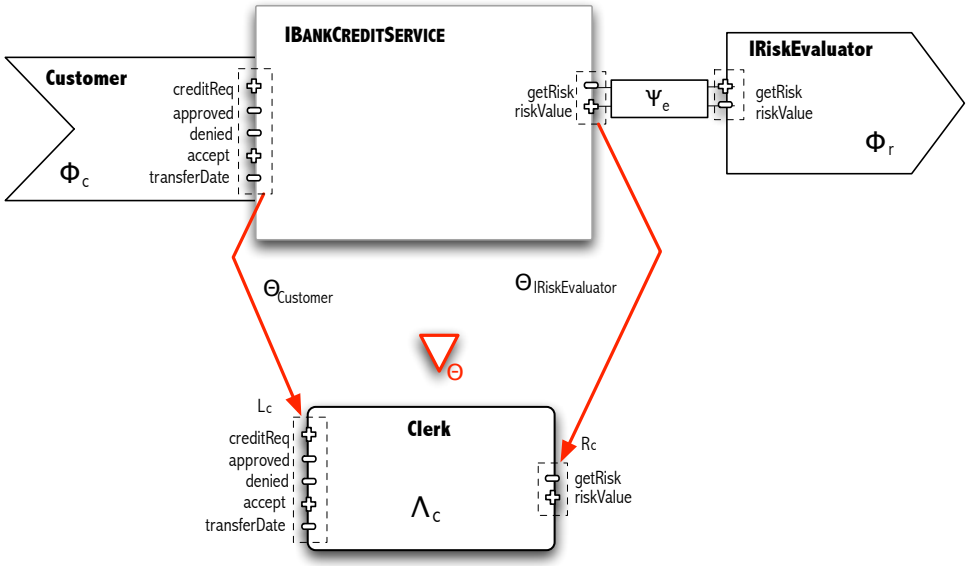

Figure 7: Example of an orchestration: $\nu_{Clerk} \lhd_\theta$ IBANKCREDITSERVICE.

The following result clarifies what we mean by 'canonical' and justifies the definition:

Theorem 3.9 (Canonicity of orchestrations)**.** *Let $\langle \alpha, \theta \rangle$ be an orchestration of an interface $i = \langle i^{\rightarrow}, I^{\leftarrow}, M^i, \Phi^i, \Psi^i \rangle$. For every requires-point r of $I^{\leftarrow}$, let*

- *β_r be an ARN with an interaction point $\langle q_r, M'_r \rangle$ such that $P_{\beta_r, q_r, M'_r} \vDash_{\theta'_r} \Phi_r$ where $\theta'_r : M_r \rightarrow M'_r$ is a polarity-preserving injection,*
- *w_r be a connection between α and β_r via a channel $c_r = \langle M_r, \Lambda_r \rangle$ where Λ_r validates Ψ_r and attachments θ_r, θ'_r.*

Then, the composition $\beta^* = (\alpha \; ||^{r \in I^{\leftarrow}}_{\theta(r), w_r, \langle q_r, M'_r \rangle} \beta_r)$ satisfies $P_{\beta^*, \theta(i^{\rightarrow})} \vDash_{\theta_{i^{\rightarrow}}} \Phi_{i^{\rightarrow}}$.

Proof.

- From $P_{\beta_r, q_r, M'_r} \vDash_{\theta'_r} \Phi_r$, we derive that $\beta_r \preceq_{q_{r\_} \circ \theta'_r} \alpha_r$.
- From Theo.2.29, we conclude that $\beta^* \preceq_\rho \alpha^*$ where ρ is the identity on α and coincides with $(q_{r\_} \circ \theta'_r)$ on α_r (α and α^* being as in Def. 3.8).
- From Prop. 2.31, the identity is a process morphism $P_{\alpha^*, \theta(i^{\rightarrow})} \rightarrow P_{\beta^*, \theta(i^{\rightarrow})}$.
- Because $\langle \alpha, \theta \rangle$ is an orchestration of i, we know that $P_{\alpha^*, \theta(i^{\rightarrow})} \vDash_{\theta_{i^{\rightarrow}}} \Phi_{i^{\rightarrow}}$.

- From Prop. 3.5 we can then conclude that $P_{\beta^*,\theta(i^{\rightarrow})} \models_{\theta_i\rightarrow} \Phi_{i\rightarrow}$.

□

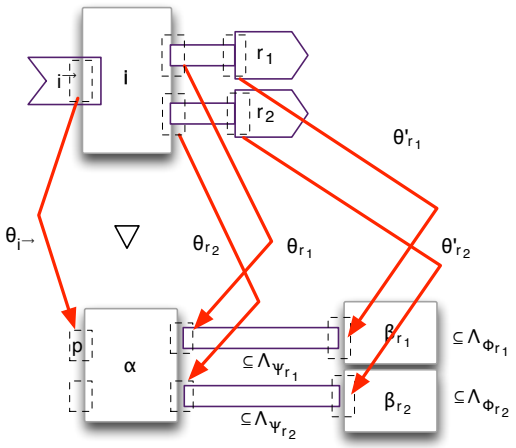

Figure 8: $\alpha \lhd_\theta i$ iff the properties offered through $\Phi_i\rightarrow$ are validated by all compositions of α with ARNs and via channels that validate the specifications of the requires-points.

Fig. 8 summarises the constructions involved in the proof. The result means that, no matter what the external services that bind to the requires-points do and how the channels transmit messages (as long as they satisfy the corresponding specifications), the ARN will be able to deliver the properties specified in its provides-point.

A property that is often useful in proofs is that the internal structure of the orchestration is not relevant:

Proposition 3.10. *Given a service interface i and an ARN α, $(\alpha \lhd_\theta i)$ iff $(\nu_{P_\alpha} \lhd_\theta i)$ where ν_{P_α} is the atomic ARN consisting of the process defined by α (cf. Def. 2.12).*

Proof. Trivial.

Requiring the ARN α^* to be consistent is also important because an interconnection that leads to an inconsistent composition would vacuously satisfy any specification (there would be no behaviours to check against the specification).

Definition 3.11 (Well-defined orchestration)**.** *An orchestration α of a service-interface i is said to be well defined if the ARN α^* as constructed in Def. 3.8 is consistent. We use $\alpha \blacktriangleleft_\theta i$ to indicate that α is a well-defined orchestration of i.*

Naturally, α itself needs to be consistent to be a well-defined orchestration.

Corollary 2.24 gives us a sufficient condition for checking that an an orchestration is well defined:

Corollary 3.12. *Given a service-interface $i = \langle i^{\rightarrow}, I^{\leftarrow}, M, \Phi, \Psi\rangle$ and an orchestration $(\alpha \lhd_\theta i)$, $(\alpha \blacktriangleleft_\theta i)$ if:*

1. *α is a safe progress-enabled ARN that is delivery-enabled in relation to the images of the requires-points.*

2. *The ARNs α_r defined by the processes $\langle\{M_r\}, \Lambda_{\Phi_r}\rangle$ are safe and delivery-enabled in relation to $\{M_r\}$.*
3. *The channels $\langle M_r, \Lambda_{\Psi_r}\rangle$ are safe and publication-enabled.*

We defer the discussion on how to ensure points 2 and 3 to Section 3.5.

Likewise, Theorem 3.9 does not guarantee by itself that the resulting ARN β^* is consistent. As above, we can guarantee consistency provided that:

1. the orchestration α is safe, progress-enabled and delivery-enabled in relation to the interaction-points that correspond to the requires-points;
2. the ARNs β_r that implement the specifications of the requires-points are consistent, safe, progress-enabled and delivery-enabled in relation to the interaction-points through which they connect to α;
3. the channels involved are safe and publication-enabled.

Another corollary concerns orchestrations defined through specifications:

Corollary 3.13. *Let $i = \langle i^{\rightarrow}, I^{\leftarrow}, M, \Phi, \Psi\rangle$ be a service-interface such that all ports are mutually disjoint. Let $\langle A, \Gamma\rangle$ be a specification such that its alphabet A includes $A_{M_{i\rightarrow}}$ and all the A_{M_r} where $r{\in}I^{\leftarrow}$. Let $i_\alpha = \langle\{M_r^{op} : r{\in}I^{\leftarrow}\}{\cup}\{M_{i\rightarrow}\}, \Lambda_\Gamma\rangle$ be the ARN consisting of a single process whose ports are those of the interface-points and whose behaviour is generated by the set Γ of sentences (cf. Def 3.1).*

- *If $(\Gamma \cup \bigcup_{r\in I^{\leftarrow}} \Phi_r \cup \bigcup_{r\in I^{\leftarrow}} \Psi_r) \models \Phi_{i\rightarrow}$ then $(i_\alpha \lhd_{id} i)$ (i_α is an orchestration of i).*
- *If in addition ($\Gamma \cup \bigcup_{r\in I^{\leftarrow}} \Phi_r \cup \bigcup_{r\in I^{\leftarrow}} \Psi_r$) is consistent, $(i_\alpha \blacktriangleleft_{id} i)$.*

3.3. Composition of service interfaces

We now turn our attention to the composition of interfaces, an essential ingredient of any interface algebra.

Definition 3.14 (Match)**.** *A match between two interfaces $i = \langle i^{\rightarrow}, I^{\leftarrow}, M^i, \Phi^i, \Psi^i\rangle$ and $j = \langle j^{\rightarrow}, J^{\leftarrow}, M^j, \Phi^j, \Psi^j\rangle$ is a pair $\langle r, \delta\rangle$ where $r{\in}I^{\leftarrow}$ and $\delta{:}M_r^i{\rightarrow}M_{j\rightarrow}^j$ is a polarity-preserving injection such that $\Phi_{j\rightarrow}^j \models \delta(\Phi_r^i)$. Two interfaces are said to be compatible if their sets of interface-points are disjoint and admit a match.*

That is, a match maps a requires-point of one of the interfaces to the provides-point of the other in such a way that the required properties are entailed by the provided ones. Notice that, because the identity of the interface-points is immaterial, requiring that the sets of points of the interfaces be disjoint is not restrictive at all. We typically use $\delta{:}r$ to refer to a match.

Definition 3.15 (Composition of interfaces)**.** *Given a match $\delta{:}r$ between compatible interfaces i and j, their composition $(i \parallel_{\delta:r} j)$ is $\langle i^{\rightarrow}, K^{\leftarrow}, M, \Phi, \Psi\rangle$ where:*

- $K^{\leftarrow} = J^{\leftarrow} \cup (I^{\leftarrow} \setminus \{r\})$.
- *$\langle M, \Phi, \Psi\rangle$ coincides with $\langle M^i, \Phi^i, \Psi^i\rangle$ and $\langle M^j, \Phi^j, \Psi^j\rangle$ on the corresponding interface-points.*

Notice that the composition of interfaces is not commutative: one of the interfaces (on the left) plays the role of client and the other (on the right) of supplier of services.

We can now prove compositionality, i.e., that the composition of the orchestrations of compatible interfaces is an orchestration of the composition of the interfaces.

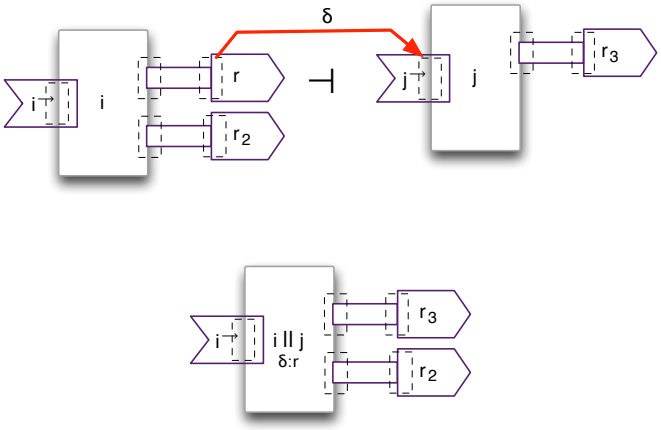

Figure 9: Match between interfaces i and j and the interface that results from their composition.

Theorem 3.16 (Composition of orchestrations)**.** *Let* $i = \langle i^{\rightarrow}, I^{\leftarrow}, M^i, \Phi^i, \Psi^i \rangle$ *and* $j = \langle j^{\rightarrow}, J^{\leftarrow}, M^j, \Phi^j, \Psi^j \rangle$ *be compatible interfaces,* $\delta{:}r$ *a match between them,* $\langle \alpha, \theta \rangle$ *and* $\langle \beta, \sigma \rangle$ *orchestrations of* i *and* j*, respectively, with disjoint graphs, and* $\langle M_r, \Lambda \rangle$ *a channel such that* $\Lambda \models \Psi^i_r$*. Then,*

$$(\alpha \bigparallel_{\theta(r),w,\sigma(j^{\rightarrow})} \beta) \lhd_{\kappa} (i \parallel_{\delta:r} j)$$

where κ *coincides with* θ *on* I *and with* σ *on* J*,* $w = \langle M_p \xleftarrow{\theta_r} M_r \xrightarrow{\sigma_{j^{\rightarrow}} \circ \delta} M_q, \Lambda \rangle$*,* $\theta(r){=}\langle p, M_p \rangle$ *and* $\sigma(j^{\rightarrow}) = \langle q, M_q \rangle$*.*

Proof. We start by noticing that κ is an injection because we assumed that α and β have disjoint graphs. Let $(\alpha \parallel \beta)^*$ be the ARN constructed as in Def. 3.8

$$((\alpha \bigparallel_{\langle p,M_p \rangle,w,\langle q,M_q \rangle} \beta) \bigparallel_{\kappa(t),w_t,\langle t,M_t \rangle}^{t \in K^{\leftarrow}} \gamma_t)$$

i.e., the ARNs γ_t are the models of the specifications of the requires-points $t \in K^{\leftarrow}$. We now prove that $(\alpha \parallel \beta)^*$ validates the provides-point:

- Because $\langle \beta, \sigma \rangle$ is an orchestration of j, $P_{\beta^*,\sigma(j^{\rightarrow})} \vDash_{\sigma_{j^{\rightarrow}}} \Phi^j_{j^{\rightarrow}}$.
- We also know that, because $\delta{:}r$ is a match, $\Phi^j_{j^{\rightarrow}} \models \delta(\Phi^i_r)$.

  Therefore, $P_{\beta^*,\sigma(j^{\rightarrow})} \vDash_{\delta \circ \sigma_{j^{\rightarrow}}} \Phi^i_r$.
- By applying Theo. 3.9 to α and the family β_t, $t{\in}I^{\leftarrow}$, of ARNs defined by
  - $\beta_r = \beta^* = (\beta \|^{t \in J^{\leftarrow}}_{\sigma(t),w_t,\langle t,M_t \rangle} \gamma_t)$
  - $\beta_t = \gamma_t$ for $t \neq r$

  we can conclude that $P_{(\alpha \| \beta)^*,\theta(i^{\rightarrow})} \vDash_{\theta_{i^{\rightarrow}}} \Phi^i_{i^{\rightarrow}}$

□

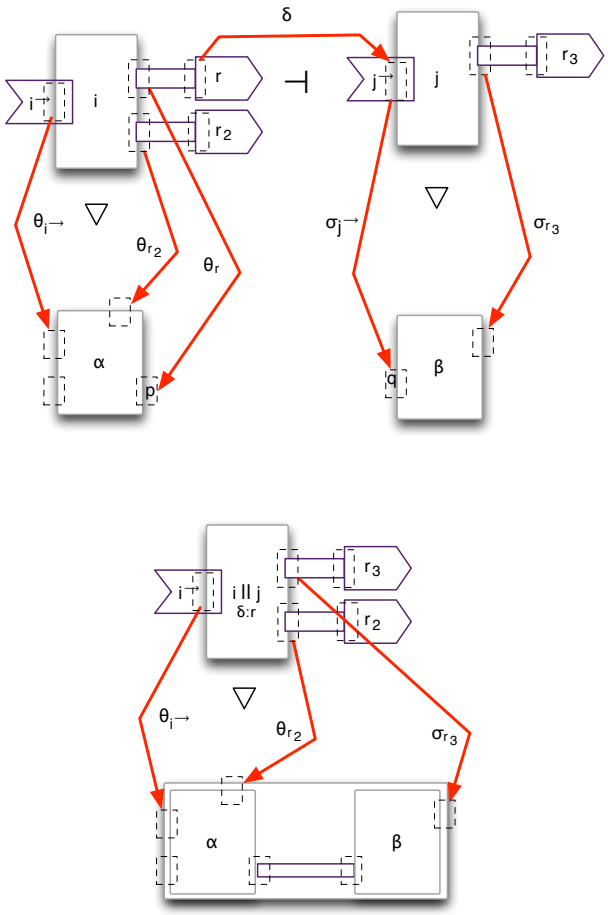

Figure 10: The context of Theorem 3.16.

Notice that the proof essentially rearranges the ARN $(\alpha \parallel \beta)^*$ as a composition of α with the models of the specifications of its requires-points except for r where we use β^*, which we know is a model of Φ_r.

Compositionality is one of the key properties required in [18] for a suitable notion of interface. From the software engineering point of view, it means that there is indeed a separation between interfaces and their implementations in the sense that composition can be performed at the interface level independently of the way the interfaces will be implemented.

Well-definedness of the composition can be guaranteed as follows:

Theorem 3.17 (Composition of orchestrations)**.** *In the circumstances of Theorem 3.16, if*

- $(\alpha \blacktriangleleft_\theta i)$ *and* $(\beta \blacktriangleleft_\sigma j)$,
- α *is safe, progress-enabled and delivery-enabled in relation to* $\theta(r)$,
- β *is safe, progress-enabled and delivery-enabled in relation to* $\sigma(j^{\rightarrow})$,
- $\langle M_r, \Lambda_{\Psi^i_{M_r}} \rangle$ *is publication-enabled*

then

$$(\alpha \underset{\theta(r),w,\sigma(j^{\rightarrow})}{||} \beta) \blacktriangleleft_{\kappa} (i \parallel_{\delta:r} j)$$

It follows from this result that, if a match is established between the interface of a service i and the interface of another service j, any orchestrations of i and j that fulfil the stated conditions will work properly together at run time (i.e., there will be no interaction errors).

In Fig. 11 we illustrate the composition of the orchestrations

$$\nu_{Clerk} \lhd_{\theta} \text{IBANKCREDITSERVICE}$$

$$\nu_{RiskEvaluator} \lhd_{\kappa} \text{IRISKEVALSERVICE}$$

where IRISKEVALSERVICE is an interface with provides-point $RECustomer$ whose set of properties Φ_e includes $\Box(request¡ \supset \Diamond_{\leq 3} result!)$. This interface is orchestrated by the atomic ARN $\nu_{RiskEvaluator}$ presented before, $\kappa_{RECustomer}$ being the identity.

The matching between the two interfaces is established by the polarity preserving mapping δ: $getRisk \mapsto request$, $riskValue \mapsto result$. The requirement in Φ_r translates through δ to $\Box(request¡ \supset \Diamond_{\leq 4} result!)$, which is trivially entailed by Φ_e. For the composition of the two services, we take the channel w_{ce} also used in the ARN presented in Fig. 2, as Λ_w validates $\delta(\Psi_r)$. The result of this composition, presented at the bottom of Fig. 11, is an interface with a single interface-point orchestrated by the ARN presented before.

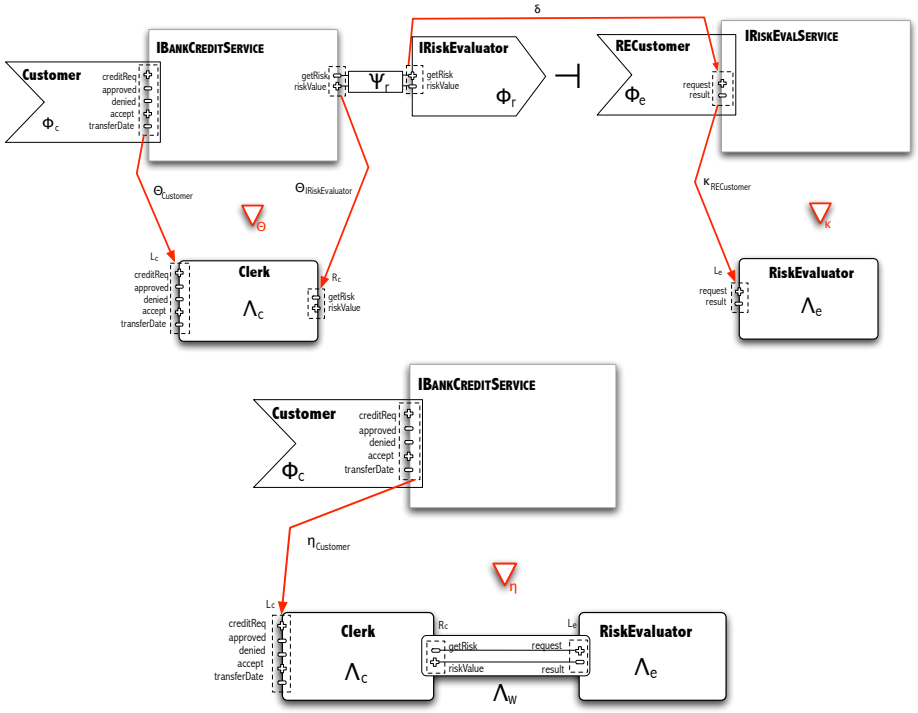

Figure 11: The composition of two orchestrations.

3.4. Refinement of service interfaces

An important ingredient of the theory of component interfaces developed in [18] is a notion of compositional refinement that applies to interfaces (for top-down design) and a notion of compositional abstraction for implementations (orchestrations in the case of services), that can support bottom-up verification. Both notions are based on a reflexive and transitive binary relation $\preceq$ that, left to right, means refinement and, right to left, means abstraction. In this section we address refinement that applies to interfaces while abstraction for orchestrations is as in Section 2.5.

Definition 3.18 (Refinement of interfaces)**.** *A refinement of* $i = \langle i^{\rightarrow}, I^{\leftarrow}, M^i, \Phi^i, \Psi^i \rangle$ *consists of*

- *an interface* $j = \langle j^{\rightarrow}, J^{\leftarrow}, M^j, \Phi^j, \Psi^j \rangle$
- *a bijection* ρ *between* I *and* J
- *a polarity-preserving injection* $\rho_{i^{\rightarrow}} : M^i_{i^{\rightarrow}} \to M^j_{j^{\rightarrow}}$
- *an* $I^{\leftarrow}$*-indexed family of polarity-preserving bijections* $\rho_r : M^j_{\rho(r)} \to M^i_r$

such that

- $\rho(i^{\rightarrow}) = j^{\rightarrow}$
- $\Phi^j_{j^{\rightarrow}} \models \rho_{i^{\rightarrow}}(\Phi^i_{i^{\rightarrow}})$,
- $\Phi^i_r \models \rho_r(\Phi^j_{\rho(r)})$ *for every* $r \in I^{\leftarrow}$,
- $\Psi^i_r \models \rho_r(\Psi^j_{\rho(r)})$ *for every* $r \in I^{\leftarrow}$.

We write $j \preceq i$ to indicate that the interface j refines i, or $j \preceq_\rho i$ when we want to make the refinement mapping $\langle \rho, \{\rho_r\}_{r \in I} \rangle$ explicit. The refinement relation $\preceq$ thus defined is reflexive and transitive.

Because ρ is a bijection, the number of interface-points of the two service interfaces are the same: none can be added or removed. This is because, on the one hand, and following the tradition of 'don't ask more', an orchestration of j cannot rely on more external services than those required by i, otherwise it would not be able to orchestrate i. On the other hand, the number of requires-points cannot decrease either: if a service interface declares a particular requires-point, this is because it wants to optimise the provision of the service by procuring an external provider at run time; therefore, refinement should preserve this decision instead of forcing that external functionality to be implemented in the orchestration.

Notice that, through refinement, new messages can be added to the provides-point and more properties can be added ('don't offer less'). On the other hand, in relation to requires-points, refinement can weaken the required properties ('don't ask more'), thus enlarging the space of providers, but it cannot change the required messages.

A fundamental property of refinement is that if j refines i, then any orchestration of j should also be an orchestration of i.

Theorem 3.19 (Refinement vs implementation)**.** *Given interfaces* i *and* j *such that* j *refines* i*, every orchestration of* j *also defines an orchestration of* i*.*

Proof. See Figure 12 for a sketch of the context of the proof. Assume that $\alpha \lhd_\theta j$ and $j \preceq_\rho i$.

1. We start by defining the mapping θ' from I to I_α:
   - $\theta'(i^{\rightarrow}) = \theta(j^{\rightarrow})$ and $\theta'_{i\rightarrow} = \theta_{j\rightarrow} \circ \rho_{i\rightarrow}$ where $j^{\rightarrow} \underline{\theta} p$.
   - For every $r \in I^{\leftarrow}$, $\theta'(r) = \theta(\rho(r))$ and $\theta'_r = \theta_{\rho(r)} \circ \rho_r^{-1} : M_r^{i^{op}} \rightarrow M_p$ where $\rho(r) \underline{\theta} p$.

   The function θ' is injective because ρ and θ are injective. Every function θ'_r is a composition of polarity-preserving injections, so it is itself a polarity-preserving injection.
2. Let $\alpha_i^* = (\alpha \parallel^{r \in I^{\leftarrow}}_{\theta(r), w_r^i, \langle r, M_r \rangle} \alpha_r^i)$ where the α_r^i and w_r^i are as in Def. 3.8.
   - Because, for every $r \in J^{\leftarrow}$, $\Phi_r^i \models \rho_r(\Phi^j_{\rho(r)})$ and $\Psi_r^i \models \rho_r(\Psi^j_{\rho(r)})$, we can conclude that, for every $r \in J^{\leftarrow}$, $\Lambda_{\Phi_r^i}|_{\rho_r} \models \Phi^j_{\rho(r)}$ and $\Lambda_{\Psi_r^i}|_{\rho_r} \models \Psi^j_{\rho(r)}$.
   - Because $\alpha \lhd_\theta j$ and ρ is a bijection between I and J, we are in the conditions of Theo. 3.9 and can conclude that $P_{\alpha_i^*, \theta(j^{\rightarrow})} \vdash_{\theta_{j\rightarrow}} \Phi^j_{j\rightarrow}$.
   - Because $\Phi^j_{j\rightarrow} \models \rho_{i\rightarrow}(\Phi^i_{i\rightarrow})$, we have that $P_{\alpha_i^*, \theta(j^{\rightarrow}) \circ \rho_{i\rightarrow}} \vdash_{\theta_{j\rightarrow} \circ \rho_{i\rightarrow}} \Phi^i_{i\rightarrow}$.
   - Given that $\theta'_{i\rightarrow} = \theta_{j\rightarrow} \circ \rho_{i\rightarrow}$, we conclude that $P_{\alpha_i^*, \theta'(i^{\rightarrow})} \vdash_{\theta'_{i\rightarrow}} \Phi^i_{i\rightarrow}$ i.e., $\alpha \lhd_{\theta'} i$.

□

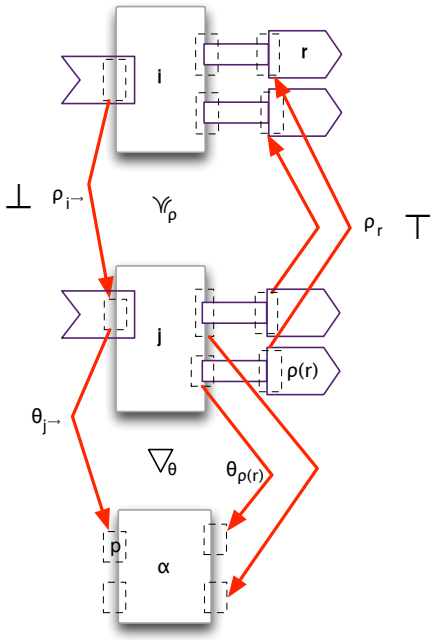

Figure 12: The context of Theorem 3.19.

Another fundamental property of refinement so that it can support top-down design (i.e., that an interface i can be refined, iteratively, into an interface $j \preceq i$) is that the elements of a composite interface can be refined independently [18], i.e., the refinement relation $\preceq$ is compositional w.r.t. $\parallel$ in the following sense:

Theorem 3.20 (Compositional refinement)**.** *Let $\delta : r$ be a match between service interfaces i_1 and i_2. If $j_1 \preceq_{\rho_1} i_1$ and $j_2 \preceq_{\rho_2} i_2$, then*

- $\rho_{2_{i_2^{\rightarrow}}} \circ \delta \circ \rho_{1_r} : \rho_1(r) \rightarrow j_2^{\rightarrow}$ *is a match between j_1 and j_2*
- $(j_1 \parallel_{\rho_{2_{i_2^{\rightarrow}}} \circ \delta \circ \rho_{1_r}} j_2) \preceq (i_1 \parallel_{\delta} i_2)$

Proof. See Fig. 13 for a sketch of the context of the proof. We start by proving that $\rho_{2_{i_2^{\rightarrow}}} \circ \delta \circ \rho_{1_r}$ is indeed a match between j_1 and j_2.

1. Because all the functions involved are polarity-preserving injections, so is their composition.
2. Because δ is a match between i_1 and i_2, we know that $\Phi_{i_2}^{i_2^{\rightarrow}} \models \delta(\Phi_{i_1}^r)$ and, by Prop. 3.2.3, $\rho_{2_{i_2^{\rightarrow}}}(\Phi_{i_2}^{i_2^{\rightarrow}}) \models \rho_{2_{i_2^{\rightarrow}}}(\delta(\Phi_{i_1}^r))$.
3. Because $j_2 \preceq_{\rho_2} i_2$, $\Phi_{j_2}^{\rho_2(i_2^{\rightarrow})} \models \rho_{2_{i_2^{\rightarrow}}}(\Phi_{i_2}^{i_2^{\rightarrow}})$ and, from (2), $\Phi_{j_2}^{\rho_2(i_2^{\rightarrow})} \models \rho_{2_{i_2^{\rightarrow}}}(\delta(\Phi_{i_1}^r))$.
4. On the other hand, $j_1 \preceq_{\rho_1} i_1$ implies that $\Phi_{i_1}^r \models \rho_{1_r}(\Phi_{j_1}^{\rho_1(r)})$, which by Prop. 3.2.3 implies $\rho_{2_{i_2^{\rightarrow}}}(\delta(\Phi_{i_1}^r)) \models \rho_{2_{i_2^{\rightarrow}}}(\delta(\rho_{1_r}(\Phi_{j_1}^{\rho_1(r)})))$
5. Finally, from (3) and (4), $\Phi_{j_2}^{\rho_2(i_2^{\rightarrow})} \models \rho_{2_{i_2^{\rightarrow}}}(\delta(\rho_{1_r}(\Phi_{j_1}^{\rho_1(r)})))$.

It remains to prove that $(j_1 \parallel_{\rho_{2_{i_2^{\rightarrow}}} \circ \delta \circ \rho_{1_r}} j_2) \preceq (i_1 \parallel_{\delta} i_2)$, which is straightforward by taking the sum of the refinement mappings ρ_1 and ρ_2, i.e., the mapping that coincides with ρ_1 on the provides-point and with ρ_n on the requires-points that remain from j_n (n=1,2). □

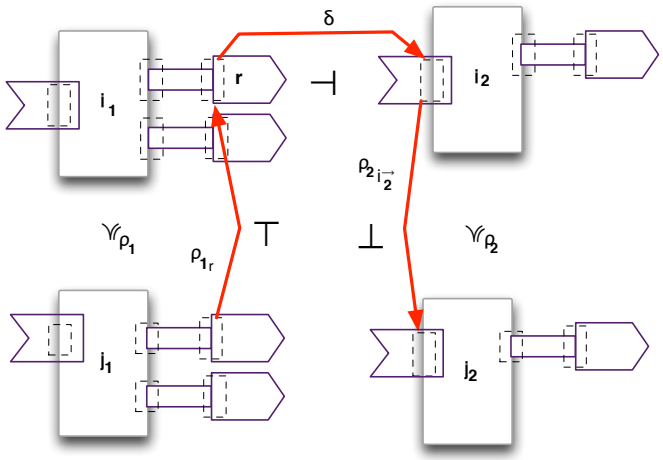

Figure 13: The context of Theorem 3.20.

Abstraction is also a way of simplifying the proof that an ARN orchestrates an interface:

Proposition 3.21 (Abstraction vs orchestration)**.** *Given an interface i and an ARN α such that $\alpha \lhd_\theta i$, if $\beta \preceq_\rho \alpha$ then $\beta \lhd_{\theta \circ \rho} i$.*

Proof. This is a simple consequence of Proposition 3.5. □

That is, to prove that an ARN orchestrates an interface, it is sufficient to prove that one of its abstractions is an orchestration of that interface. This result generalises Proposition 3.10.

3.5. Logics for service interfaces

The results of Sec. 3.2 highlight the importance of choosing specification logics so that it is possible (and effective) to check whether specifications define safe delivery-enabled processes or safe publication-enabled channels. As argued in Section 2.3, working with safety properties is justified by the fact that, within SOC, we are interested in processes whose liveness properties are bounded (bounded liveness being itself a safety property).

Several extensions of LTL have been proposed in which one can express different forms of bounded liveness. For instance, several logics for real-time systems (see [5] for an early survey) allow one to express eventuality properties of the form $\Diamond_I \phi$ where I is a time interval during which ϕ is required to become true. Another logic of interest is PROMPT-LTL [41] in which, instead of a specific bound for the waiting time, one can simply express that a sentence ϕ will become true within an unspecified bound — $\Diamond_{\mathbf{p}}\phi$. Yet another logic is PLTL [4] in which one can use variables in addition to constants to express bounds on the waiting time and reason, for example, about the existence of a bound (or of a minimal bound) for a response time.

The logic we use in this paper, which we call SAFETY-LTL, is a version of PLTL where intervals are finite and bounded by constants. It can also be seen as a restricted version of SAFETY-MTL [48] (itself a fragment of Metric Temporal Logic [40]) where, instead of an explicit model of real-time, we adopt an implicit one in which time is discrete. An advantage of adopting a discrete-time model (one in which time units are execution steps) is that we can work directly over $(2^A)^\omega$ without introducing an explicit space of real-time[3]. From a methodological point of view, the restriction can be justified by the fact that, in SOC, one often deals with 'business'-time where delays are measured in discrete time units.

Definition 3.22 (SAFETY-LTL)**.** *Let A be an alphabet.*

- *The language of* SAFETY-*LTL over A is defined by (where $a \in A$ and t is a natural number – $t \in \mathbb{N}$):*

$$\phi \quad ::= \quad a \mid \neg a \mid \phi \vee \phi \mid \phi \wedge \phi \mid \bigcirc \phi \mid \phi \,\mathcal{R}\, \phi \mid \phi \,\mathcal{R}_{\leq t}\, \phi \mid \phi \,\mathcal{U}_{\leq t}\, \phi$$

- *Sentences are interpreted over $\lambda \in (2^A)^\omega$ as follows :*

$\lambda \vDash a$ *iff* $a \in \lambda(0)$
$\lambda \vDash \neg a$ *iff* $a \notin \lambda(0)$
$\lambda \vDash \phi_1 \wedge \phi_2$ *iff* $\lambda \vDash \phi_1$ *and* $\lambda \vDash \phi_2$
$\lambda \vDash \phi_1 \vee \phi_2$ *iff* $\lambda \vDash \phi_1$ *or* $\lambda \vDash \phi_2$
$\lambda \vDash \bigcirc \phi$ *iff* $\lambda^1 \vDash \phi$
$\lambda \vDash \phi_1 \,\mathcal{R}\, \phi_2$ *iff, for all j, either* $\lambda^j \vDash \phi_2$ *or there exists* $k<j$ *s.t.* $\lambda^k \vDash \phi_1$

[3]The choice of a timed semantics – where a time sequence runs along a trace – with topological properties that allow the results given in this paper to be generalised is not trivial and is discussed in [20].

$\lambda \vDash \phi_1 \mathcal{R}_{\leq t} \phi_2$ *iff, for all* $j \leq t$*, either* $\lambda^j \vDash \phi_2$ *or there exists* $k < j$ *s.t.* $\lambda^k \vDash \phi_1$

$\lambda \vDash \phi_1 \mathcal{U}_{\leq t} \phi_2$ *iff there exists* $j \leq t$ *s.t.* $\lambda^j \vDash \phi_2$ *and, for all* $k < j$, $\lambda^k \vDash \phi_1$

- *We use the following abbreviations:*

$a \supset \phi \equiv \neg a \vee \phi$

$\Diamond_{\leq t}\phi \equiv true\, \mathcal{U}_{\leq t}\, \phi$ *— ϕ will hold before t time units (or it holds now)*

$\Box\phi \equiv false\, \mathcal{R}\, \phi$ *— now and forever ϕ*

$\Box_{\leq t}\phi \equiv false\, \mathcal{R}_{\leq t}\, \phi$ *— ϕ will hold for the next t time units (and now)*

Notice that the language is not closed under negation: negation is only available for atomic propositions (actions) and sentences are in negation normal form.

Theorem 3.23 (Safety)**.** *All the sentences of* SAFETY*-LTL express safety properties, i.e., for all sentences ϕ, Λ_ϕ is a closed set.*

Proof. By a simple induction in the structure of ϕ. □

Corollary 3.24 (Safe specifications)**.** *It follows from the previous theorem that all the specifications over* SAFETY*-LTL are safe, i.e., for all sets of sentences Φ, Λ_Φ is a safety property.*

Proof. The result follows from the fact that the intersection of any number of closed properties is closed. □

As motivated in Section 3.2, in addition to making sure that specifications of processes and channels generate safety properties, it is important that developers can guarantee that the processes thus defined are delivery-enabled in relation to their ports and that channels are publication-enabled. Ensuring delivery/publication-enabledness is not the same as proving that a process/channel satisfies a specification because those properties are not expressible as sentences whose satisfaction can be checked over individual traces: they would need to be checked over sets of traces, for which a branching-time logic would be required. In the context of a logic like SAFETY-LTL, these properties can be checked instead over the non-deterministic Bücchi automata that implement the specifications as explained in Section 2.4.

4. Related Work

In this paper, we proposed a formalisation of 'services' as interfaces for an algebra of asynchronous components inspired by the work reported in [18] on a theory of interfaces for component-based design and our previous work on a modelling language for service-oriented computing (SRML [33]). That is, we exposed and provided mathematical support for the view that services are, at a certain level of abstraction, a way of *using* software components — what is sometimes called a 'service-overlay' — and not so much a way of *constructing* software. This view is consistent with the way services are being perceived in businesses [22] and supported by architectures such as SCA [47].

In relation to SRML, this paper abstracts an interface theory for services in the style [18] from the specific semantic model and specification structures proposed in, e.g., [30]. That is, SRML offers an instance of such an interface theory and an example

of how it can be used for modelling services within a methodology that differs from the more traditional component-based approach in which components expose methods in their interfaces and bind tightly to each other (based on I/O-relations) to construct software applications. In our approach, components expose conversational, stateful interfaces through which they expose services that can be discovered by business applications or bind, on the fly, to external services. Examples of case studies for different application domains can be found in [1, 10, 11]. See also [29, 32] for a formalisation of the dynamic aspects of service discovery and binding, which can be superposed over the interface theory proposed in this paper.

Our approach also differs from assumption/guarantee (A/G) styles of specifications, which have been proposed (since [46]) for networks of processes and also used in [50] for web services. The aim of A/G is to ensure compositionality of specifications of processes by making explicit assumptions on the way they interact with their environment. The purpose of the interfaces that we propose is, instead, to specify the protocols offered to clients of the service and the protocols that the external services that the service may need to discover and bind to are required to follow. The notion of orchestration makes this clear: the purpose of the requires-points is to *create* the environment required by the orchestration to deliver the properties specified at the provides-point. This is also why our notion of composition of interfaces is not symmetric: composition of service interfaces reflects the provision of some of the services required by one interface (the client) in terms of another interface (the provider). The notion of composition of A/G specifications is meant instead to reflect the parallel composition of components and, therefore, is symmetric. In SOC, there is also a notion of composition (or aggregation) of services through which more complex services can be provided. In our approach [30], this is achieved by orchestrating the interactions between the component services and defining a service-interface for the composition that offers the properties that result from the composition in its provides-interface.

Several formal frameworks have been proposed for SOC, many of which in the form of process calculi (e.g., [12, 14, 16, 21]) or automata-based models (e.g., [8, 15, 35] for asynchronous models) that address type-theoretical aspects of service contracts. Such contracts apply to the peers involved in a service choreography – the behaviour that peers are expected to implement in order to successfully engage with the others in a choreography – and, as such, are not interfaces for service discovery and binding through provides/requires-points as we addressed in the paper. Indeed, choreography models are inherently different from ours in the sense that they study different problems: the adoption of automata reflects the need to study the properties and realisability of conversation protocols captured as words of a language of message exchange. It would be tempting to draw a parallel between their notion of composite service — a network of machines — and our ARNs, but they are actually poles apart: our aim has not been to model the conversations that characterise the global behaviour of the peers that deliver a service, but to model the network of processes executed by an *individual* peer and how that network orchestrates a service interface for that peer — that is, our approach is *orchestration-based*. Therefore, we do not make direct usage of automata, although a reification of our processes could naturally be given in terms of automata.

Different approaches to formalising the compatibility relation between clients and service providers can also be found in the literature, which are essentially process-based (e.g., [17] based on CCS and [49] based on Petri-nets). Our usage of logic for specifying required and provided properties has the advantage of being more abstract and adopt a more general model of asynchronous communication in which channels are first-class entities (reflecting the importance that they have in SOC).

Another notion of web service interface has been proposed in [9]. This work presents a specific language, not a general approach like we did in this paper, but there are some more fundamental differences between them, such as the fact that their underlying model of interaction is synchronous (method invocation): as argued in [39], web-service composition languages such BPEL (the Business Process Execution Language [52]) rely on an (asynchronous) message-passing model, which is more adequate for interactions that need to run in a loosely-coupled operating environment. The underlying approach is, like ours, orchestration-based but, once again, more specific than ours in that orchestrations are modelled through a specific class of automata supporting a restricted language of temporal logic specifications. Another fundamental difference is that, whereas in [9] the orchestration of a service is provided by an automaton, ours is provided by a network of processes (as in SCA), which provides a better model for capturing the dynamic aspects of SOC that arise from run-time discovery and binding [26]: our notion of composition is not for integration (as in CBD) but for dynamic interconnection of processes. This is also reflected in the notion of interface: the interfaces used in [9] are meant for design-time composition, the client being statically bound to the invoked service (which is the same for all invocations); the interfaces that we proposed address a different form of composition in which the provider (the "need-fulfilment mechanism") is procured at run time and, therefore, can differ from one invocation to the next, as formalised in [32] in a more general algebraic setting.

Being based on a specific language, [9] explores a number of important issues related to compatibility and consistency that arise naturally in service design when one considers semantically-rich interactions, e.g., when messages carry data or are correlated according to given business protocols. A similar orchestration-based approach has been presented in [7], which is also synchronous and based on finite-state machines, and also addresses notions of compatibility and composition of conversation protocols (though, interestingly, based on branching time). We are studying an extension of our framework that can support such richer models of interaction (and the compatibility issues that they raise), for which we are using, as a starting point, the model that we adopted in the language SRML [33], which has the advantage of being asynchronous.

With respect to aspects related to consistency of composition, which occupy a central part of our work, several notions of compatibility have been studied aimed at ensuring that services are composable, mostly in the context of process-oriented models such as automata, labelled transition systems or Petri-Nets. Compatibility in this context may have several different meanings. For example, [45] addresses the problem of ensuring that, at service-discovery time, requirements placed by a requester service are matched by the discovered services — the requirements of the requester are formulated in terms of a graph-based model of a protocol that needs to be simulated by the BPEL orchestration of any provided service that can be discovered. That is, compatibility is checked over implementations. However, one has to assume that the requester has formulated its requirements in such a way that, once bound to a discovered service that meets the requirements, its implementation will effectively work together with that of the provided service in a consistent way — a problem not addressed in that paper.

A different approach is proposed in [9] where compatibility is tested over the interfaces of services (not their implementations), which is simpler and more likely to be effective because a good interface should hide (complex) information that is not relevant for compatibility. A limitation of that approach is, as already mentioned, that it is based on a (synchronous) method-invocation model of interaction. On the other hand, the notions of interface that are proposed in [9] do not clearly distinguish between interfaces for clients of the service and interfaces for providers of required external ser-

vices, i.e., the approach is not formulated in the context of run-time service discovery and binding. Furthermore, [9] does not propose a model of composition of implementations (what is called a component algebra in [18]) so one has to assume that implementations of services with compatible interfaces, when composed, are 'consistent'. Our model formulates the notion of consistency at the level of the component algebra in a way that one can ensure, at design time, that matching required with provided services at the interface level leads to a consistent implementation of the composite service when binding the implementations of the requester and the provider services.

5. Concluding Remarks

5.1. Summary

In this paper, we have put forward a component and interface algebras for service-oriented computing (SOC) inspired by the seminal work of de Alfaro and Henzinger [18] and the service-component architecture (SCA) [47]:

The component algebra. Components in our framework are asynchronous relational nets (ARNs) consisting of processes (sets of infinite traces over an alphabet of messages) interconnected via asynchronous channels. Two operations were defined: composition and abstraction. Composition takes two ARNs and a collection of channels that connect pairs of interaction points, one from each ARN. Abstraction maps ARNs to processes by forgetting their internal structure, which we showed to be compositional in relation to the composition operation.

In this setting, we discussed the problem of ensuring that the composition of consistent ARNs (in the sense that they admit a trace that is projected to the behaviours of every process and channel) is itself consistent. This is a non-trivial problem, especially if we want to be able to check consistency in a compositional way, i.e., based only on properties of the individual ARNs and the channels used to interconnect them. This form of compositionality is required to be able to ensure consistency at design time (i.e., when the participating processes and channels are specified and implemented), which is essential in the context of SOC for supporting run-time discovery and binding (composition) of services.

We characterised a subclass of ARNs for which an answer to this problem can be provided: those that are both safe (in the sense that their processes and channels implement safety properties) and progress-enabled (in the sense that every finite joint trace can be extended with a joint action). We proved that safe progress-enabled ARNs are consistent – Theo. 2.17 – and closed under composition provided that interconnections are made through channels that are publication-enabled (i.e., channels that do not refuse the publication of messages by the processes) and over interaction-points in relation to which the ARNs are delivery-enabled (i.e., processes that do not refuse the delivery of messages by the channels) – Cor. 2.24.

Given that individual processes are always progress-enabled, all that remains in order to ensure consistency of composition is to work with safe processes and channels and check for publication/delivery-enabledness. This can be done at design time over the implementations of the channels and the processes, the complexity of which we discussed for closed (safe) reduced non-deterministic Bücchi automata (NBAs).

The interface algebra. A service interface consists of a provides-point (offering properties to customers) and a collection of requires-points, each of which specifies the properties of an external service that may be required and of the channel through which

the external service will be connected. At the level of interfaces, properties (offered and required) are specified through logical formulas.

Two operations were defined: composition and refinement. Composition takes two interfaces and a match between a requires-point of one of the interfaces (the client) and the provides-point of the other (the supplier) – the properties provided by the supplier entail those required by the client. Refinement strengthens the properties offered at the provides-point and weakens those of the requires-points. Refinement was shown to be compositional in relation to the composition operation.

A notion of orchestration of a service-interface by an ARN was also defined and shown to be compositional in relation to both the composition and refinement of interfaces. We then discussed the problem of ensuring that orchestrations are well defined, i.e., that when interconnected with orchestrations of required services, the resulting composition is consistent and delivers the properties offered at the provides-point. Using the results obtained for ARN composition, we are able to ensure consistency provided that orchestrations are safe, progress-enabled, and delivery-enabled in relation to the requires-points, and that the specifications of the channels and external services given at the requires-points denote channels and processes that are safe and either publication-enabled (in the case of the channels) or delivery-enabled (in the case of the processes) – Cor. 3.12.

It then remained to discuss how to guarantee that the specifications given at the requires-points denote safe processes and channels. For that purpose, we presented a fragment of linear temporal logic – SAFETY-LTL – in which all the behaviours that can be specified are safety properties. Closed reduced NBAs can be used as implementations of such specifications. Because checking processes/channels for delivery/publication enabledness can be done at design time (i.e., when implementations are chosen for orchestrating service interfaces) over those automata, there is no need for any additional checking to be made at discovery/run time to guarantee consistency; the only checking that needs to be made at run time is that the specifications of provides-points entail the specifications of the corresponding requires-points. Naturally, these are all sufficient conditions.

5.2. Further work

A question that arises from the work that we have presented is whether it can be generalised, either to other models of behaviour or specification logics. For example, and although justification can be (and was) given for working with the implicit model of time enforced by SAFETY-LTL, application domains in which timing requirements are more critical (e.g., finance) would benefit from using an explicit model of time based on the real numbers. Logics such as SAFETY-MTL [48] could still be used over such a domain in order to restrict behaviours to safety properties. Probabilistic models have also been emerging as providing useful ways of addressing behaviour of services that depend on properties that, such as resource availability or performance, are intrinsically stochastic. What is not clear is if (and how) one can also generalise the characterisation of ARNs for which consistency can be ensured. For instance, the proof of Theo. 2.17 relies on properties of trace semantics (namely finite branching) that do not generalise immediately to a real-time or probabilistic domain. One point that we intend to investigate further concerns, indeed, the interplay between consistency, safety, the behavioural model and the associated logic. Frameworks such as institutions [36] could provide a starting point, though extensions are clearly required in order to have a finer characterisation of the topological properties that are required of behaviour models to address the kind of properties discussed herein.

Other lines for further work concern the dynamic aspects that are intrinsic to SOC in virtue of the run-time discovery, selection and binding processes. We plan to use, as a starting point, the algebraic semantics that we developed for SRML [32]. Important challenges that arise here relate to the unbounded nature of the configurations (ARNs) that execute business applications in a service-oriented setting, which is quite different from the complexity of the processes and communication channels that execute in those configurations.

Acknowledgments

We would like to thank Nir Piterman, Emilio Tuosto and the reviewers of [27] and [28] for many helpful comments and suggestions.

References

[1] J. Abreu, L. Bocchi, J. L. Fiadeiro, and A. Lopes. Specifying and composing interaction protocols for service-oriented system modelling. In J. Derrick and J. Vain, editors, *FORTE*, volume 4574 of *LNCS*, pages 358–373. Springer, 2007.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[3] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[4] R. Alur, K. Etessami, S. L. Torre, and D. Peled. Parametric temporal logic for "model measuring". *ACM Trans. Comput. Log.*, 2(3):388–407, 2001.

[5] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *REX Workshop*, volume 600 of *LNCS*, pages 74–106. Springer, 1991.

[6] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[7] B. Benatallah, F. Casati, and F. Toumani. Representing, analysing and managing web service protocols. *Data Knowl. Eng.*, 58(3):327–357, 2006.

[8] A. Betin-Can, T. Bultan, and X. Fu. Design for verification for asynchronously communicating web services. In Ellis and Hagino [23], pages 750–759.

[9] D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web service interfaces. In Ellis and Hagino [23], pages 148–159.

[10] L. Bocchi, J. L. Fiadeiro, and A. Lopes. Service-oriented modelling of automotive systems. In *COMPSAC*, pages 1059–1064. IEEE Computer Society, 2008.

[11] L. Bocchi, J. L. Fiadeiro, and A. Lopes. A use-case driven approach to formal service-oriented modelling. In T. Margaria and B. Steffen, editors, *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 155–169. Springer, 2008.

[12] M. Boreale et al. SCC: A service centered calculus. In M. Bravetti, M. Núñez, and G. Zavattaro, editors, *WS-FM*, volume 4184 of *LNCS*, pages 38–57. Springer, 2006.

[13] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[14] M. Bravetti and G. Zavattaro. A foundational theory of contracts for multi-party service composition. *Fundam. Inform.*, 89(4):451–478, 2008.

[15] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW*, pages 403–410, 2003.

[16] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In De Nicola [19], pages 2–17.

[17] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. *ACM Trans. Program. Lang. Syst.*, 31(5), 2009.

[18] L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In T. A. Henzinger and C. M. Kirsch, editors, *EMSOFT*, volume 2211 of *LNCS*, pages 148–165. Springer, 2001.

[19] R. De Nicola, editor. *Programming Languages and Systems*, volume 4421 of *LNCS*. Springer, 2007.

[20] B. Delahaye, J. L. Fiadeiro, A. Legay, and A. Lopes. A timed component algebra for services. In D. Beyer and M. Boreale, editors, *FORTE*, LNCS. Springer, 2013.

[21] P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In H. Seidl, editor, *ESOP*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.

[22] A. Elfatatry. Dealing with change: components versus services. *Commun. ACM*, 50(8):35–39, 2007.

[23] A. Ellis and T. Hagino, editors. *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*. ACM, 2005.

[24] J. L. Fiadeiro. *Categories for Software Engineering*. Springer, 2004.

[25] J. L. Fiadeiro. Designing for software's social complexity. *IEEE Computer*, 40(1):34–39, 2007.

[26] J. L. Fiadeiro and A. Lopes. A model for dynamic reconfiguration in service-oriented architectures. In M. A. Babar and I. Gorton, editors, *ECSA*, volume 6285 of *LNCS*, pages 70–85. Springer, 2010.

[27] J. L. Fiadeiro and A. Lopes. An interface theory for service-oriented design. In D. Giannakopoulou and F. Orejas, editors, *FASE*, volume 6603 of *LNCS*, pages 18–33. Springer, 2011.

[28] J. L. Fiadeiro and A. Lopes. Consistency of service composition. In J. de Lara and A. Zisman, editors, *FASE*, volume 7212 of *LNCS*, pages 63–77. Springer, 2012.

[29] J. L. Fiadeiro and A. Lopes. A model for dynamic reconfiguration in service-oriented architectures. *Software and Systems Modeling*, pages 1–19, 2012.

[30] J. L. Fiadeiro, A. Lopes, and J. Abreu. A formal model for service-oriented interactions. *Sci. Comput. Program.*, 77(5):577–608, 2012.

[31] J. L. Fiadeiro, A. Lopes, and L. Bocchi. Algebraic semantics of service component modules. In J. L. Fiadeiro and P.-Y. Schobbens, editors, *WADT*, volume 4409 of *LNCS*, pages 37–55. Springer, 2006.

[32] J. L. Fiadeiro, A. Lopes, and L. Bocchi. An abstract model of service discovery and binding. *Formal Asp. Comput.*, 23(4):433–463, 2011.

[33] J. L. Fiadeiro, A. Lopes, L. Bocchi, and J. Abreu. The SENSORIA reference modelling language. In Wirsing and Hölzl [55], pages 61–114.

[34] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, editors, *WWW*, pages 621–630. ACM, 2004.

[35] X. Fu, T. Bultan, and J. Su. Conversation protocols: a formalism for specification and verification of reactive electronic services. *Theor. Comput. Sci.*, 328(1-2):19–37, 2004.

[36] J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[37] M. G. Gouda. Closed covers: To verify progress for communicating finite state machines. *IEEE Trans. Software Eng.*, 10(6):846–855, 1984.

[38] M. G. Gouda, E. G. Manning, and Y.-T. Yu. On the progress of communications between two finite state machines. *Information and Control*, 63(3):200–216, 1984.

[39] R. Kazhamiakin, M. Pistore, and L. Santuari. Analysis of communication models in web service compositions. In L. Carr, D. D. Roure, A. Iyengar, C. A. Goble, and M. Dahlin, editors, *WWW*, pages 267–276. ACM, 2006.

[40] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[41] O. Kupferman, N. Piterman, and M. Y. Vardi. From liveness to promptness. *Formal Methods in System Design*, 34(2):83–103, 2009.

[42] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[43] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In F. B. Schneider, editor, *PODC*, pages 137–151. ACM, 1987.

[44] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.

[45] A. Martens. Process oriented discovery of business partners. In C.-S. Chen, J. Filipe, I. Seruca, and J. Cordeiro, editors, *ICEIS (3)*, pages 57–64, 2005.

[46] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.

[47] OSOA. Service component architecture: Building systems using a service oriented architecture, 2005. White paper available from www.osoa.org.

[48] J. Ouaknine and J. Worrell. Safety metric temporal logic is fully decidable. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920 of *LNCS*, pages 411–425. Springer, 2006.

[49] W. Reisig. Towards a theory of services. In R. Kaschek, C. Kop, C. Steinberger, and G. Fliedl, editors, *UNISCON*, volume 5 of *LNBIP*, pages 271–281. Springer, 2008.

[50] M. Solanki, A. Cau, and H. Zedan. Introducing compositionality in web service descriptions. In *FTDCS*, pages 14–20. IEEE Computer Society, 2004.

[51] J. Su, T. Bultan, X. Fu, and X. Zhao. Towards a theory of web service choreographies. In M. Dumas and R. Heckel, editors, *WS-FM*, volume 4937 of *LNCS*, pages 1–16. Springer, 2007.

[52] O. W. TC. Web services business process execution language, 2007. Version 2.0. Technical report, OASIS.

[53] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 332–344. IEEE Computer Society, 1986.

[54] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.

[55] M. Wirsing and M. M. Hölzl, editors. *Rigorous Software Engineering for Service-Oriented Computing*, volume 6582 of *LNCS*. Springer, 2011.